

Programming Language Concepts

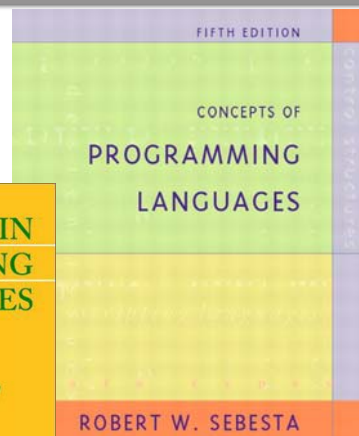
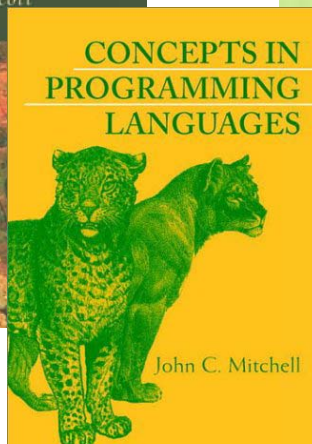
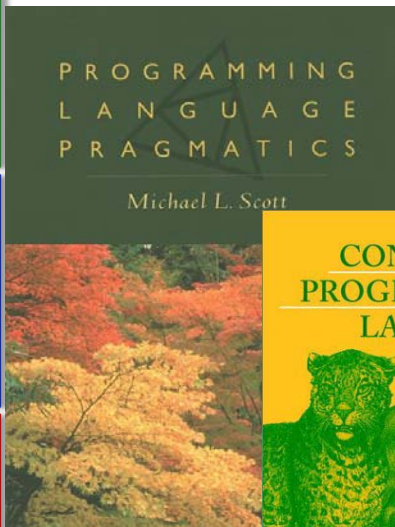
Klaus Ostermann

Darmstadt University of Technology
Software Technology Group
ostermann@informatik.tu-darmstadt.de

Getting Started

- Find lectures, links etc. at:
 - www.st.informatik.tu-darmstadt.de
 - Go to "Teaching→Lectures→Programming Language Concepts"
- Mailing list for the lecture:
 - plss03@st.informatik.tu-darmstadt.de
 - Please subscribe:
 - send an email to plss03-request@st.informatik.tu-darmstadt.de with the following body: „subscribe <your-email-address>“

- Prüfungsmöglichkeiten
 - zusammen mit anderen Vorlesungen der Fachgruppe Softwaretechnologie im Bereich Inf 2
 - oder nach Absprache
- Studien-, Diplomarbeiten
 - nach Absprache



Which programming languages do you know?

Schedule for first 3 lectures

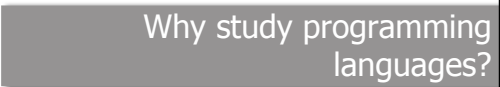

- Why study programming languages?
- Evaluating Languages
- Programming Languages and Computability
- Introduction to LISP / Scheme
- Describing Syntax and Semantics
- Names, Scopes, Bindings
- Expressions and Assignments
- Structuring Control Flow

- Potential further topics (to be determined)
 - Type Systems, Data Abstraction, Modules, OOPL, Concurrency



Intro: Why study programming languages

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de



Why study programming languages?

- Increase ability to reason about problems
 - Programs = Models that run!
 - “Think” in Java/C++/Pascal/LISP/...
 - Depth at which we can think is influenced by expressive power of the language in which we communicate our thoughts
 - Knowing more PL concepts helps us to create better models
 - even if some concepts are not supported in used PL
 - “Patterns”

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

- Improved Background for choosing appropriate languages
 - Languages that are particularly good for
 - rapid prototyping, e.g., Smalltalk
 - planes and nuclear plants, e.g., Ada
 - database access, e.g., SQL, MS Access
 - mathematicians and AI ,e.g, LISP, ML, Haskell
 - interacting with OS, "glueing", e.g., Script/Shell languages
 - domain-specific languages, e.g., VLSI, CAD, Concurrency
 - systems programming, e.g., C

- Increased ability to learn and assess new languages
 - No need to be afraid of "buzzwords"
 - Frequently "new" PL features are old concepts in a new wrapping
 - e.g, delegates in C# vs. function pointers in C++, first-class functions in LISP, blocks in Smalltalk etc.
 - Assessing a language A in comparison to language B
 - what is the "abstract" difference between Java and Smalltalk, i.e., besides syntactical differences

- Better Understanding of PL semantics in terms of PL implementation
 - Design decisions frequently influenced by implementation considerations
 - e.g. arrays in Java, primitive types in Java
 - Precise knowledge of language semantics
 - e.g., difference between equality and identity, organization of multi-dimensional arrays
 - Finding subtle bugs
 - e.g., short circuit evaluation of boolean expressions
 - Identifying performance bottlenecks
 - e.g., late binding vs. static binding, recursion vs. iteration, multi-thread vs. multi-process

- Ability to design and implement new languages
 - The interpreter of a language is just another program!
 - Mastering the idea of an interpreter is a source of great power!
 - Not only relevant to language researchers
 - Many programs contain interpreters for special-purpose languages
 - graphical description language in CAD systems
 - all sorts of configuration files, e.g., make files
 - XML hype
 - All kinds of source-code processing and analysis
 - e.g. embedded SQL, Makros, Metrics



Evaluating Languages



Evaluating Languages

- **Simplicity and Orthogonality**
 - Languages should be as simple as possible
 - otherwise programmers use only a subset of the features
 - Ada was killed by too many features
 - On the other hand, language should be powerful
 - Good measure: Orthogonality
 - a language feature is orthogonal to the other features if
 - it cannot be easily simulated by other features and
 - can be freely combined with other features, no “except” rules
 - don't confuse non-orthogonality with “syntactic sugar”
 - i.e., simple abbreviations for more complicated expressions
 - e.g., “count++” is syntactic sugar for “count = count + 1” in Java
 - Examples for non-orthogonal language features
 - Arrays in Pascal and C cannot be return type of functions
 - `\verbatim` environment in LaTeX illegal as command argument, inside minipages etc.

- Control Structures

- “Goto considered harmful”, Dijkstra 1968
- Now: more abstract control structures
 - `while ...`
 - `for`
 - `repeat ... until, do ... while`
 - `switch / case`
- Examples for even more advanced control structures
 - type-safe type casing
“`typecase point of ColorPoint do point.getColor();`”
 - exceptions: `do ... catch ... finally ...`
 - Aspect-Oriented Programming
`after (call (java.io.File.*)) { ioAccessCounter++; }`

- Structuring Data and Behavior

- “meaningful” types
 - `sum_is_too_big = 1` VERSUS `sum_is_too_big = true`
- fixed set of data types versus user defined types
 - `color = 3` versus `color = red`
- grouping related data
 - `name: String; firstname: String; age: integer, ...`
versus
`struct Person {`
 `name: String;`
 `firstname: String;`
 `...`
`}`
- incremental encoding of related data structures
- combining data and behavior → Object-Orientation
- all sorts of module constructs

- Abstraction and Information Hiding

- Use complicated structures or operations without knowing many details
- Ability to parameterize a structure or operation
- e.g., functional abstraction

abstract over function parameters:

```
int sum(int a, int b) { return a+b; }
```

parameterize function, no need to know details:

```
int mysum = sum(3,8);
```

- other kinds of abstraction, information hiding
 - type abstraction, e.g., templates, generics
 - module concept: internal and exported parts
 - visibility, e.g., private/protected/public

- Ability to reason about program behavior at compile time

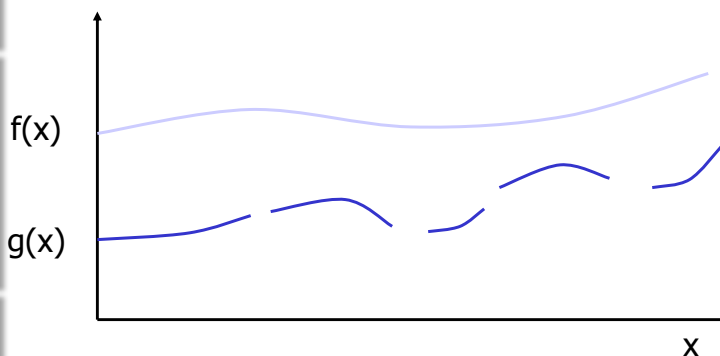
- Impossible to predict behavior in general (Haltingproblem)
- but possible to rule out large class of errors before it is too late
 - Type Checking
 - Detecting uninitialized/unused variables
 - Detecting dead code
- Control- and Data-Flow Analysis
- All kinds of proofs, partial interpretation
- Affected by many design choices
 - Aliasing
 - i.e., having multiple references to the same memory cell
 - Static Typing vs. Dynamic Typing
 - Side Effects

- What do you think of the following C function?

```
void stringcopy(Char *from, Char *to) {  
    while (*to++ = *from++);  
}
```

- Frequently desirable to prove properties or predict behavior of programs (*abstract interpretation*)
 - Is the program type safe?
 - Termination of while-loops, existence of closed loops
 - Time/space requirements
 - Initialization of variables
 - Control flow/data flow prediction
 - Security properties
 - ...
- Such questions can be formulated as mathematical (partial) functions, e.g.,
typesafe: Program $\rightarrow \{true, false\}$
- Gödel: Sufficient to consider functions on integers

Functions and Graphs



- Graph of $f = \{ \langle x, y \rangle \mid y = f(x) \}$
- Graph of $g = \{ \langle x, y \rangle \mid y = g(x) \}$

Mathematics: a function is a set of ordered pairs (graph of function)

Partial and Total Functions

- Total function $f:A \rightarrow B$ is a subset $f \subseteq A \times B$ with
 - For every $x \in A$, there is some $y \in B$ with $\langle x, y \rangle \in f$ (total)
 - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$ (single-valued)
- Partial function $f:A \rightarrow B$ is a subset $f \subseteq A \times B$ with
 - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$ (single-valued)
- Programs define partial functions for two reasons
 - partial operations (like division)
 - nontermination
$$f(x) = \text{if } x=0 \text{ then } 1 \text{ else } f(x-2)$$

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Computability

- Definition
 - A function f is **computable** if there is a program P that computes f , i.e., for any input x , the computation $P(x)$ halts with output $f(x)$
- Computability theory: It does not matter which general purpose PL we use, they are all equally powerful (Turing completeness)

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Halting function

- Decide whether program halts on input
 - Given program P and input x to P,

$$\text{Halt}(P,x) = \begin{cases} \text{yes} & \text{if } P(x) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$

Clarifications

Assume program P requires one string input x
Write P(x) for output of P when run in input x
Program P is string input to *Halt*

Fact: There is no program for *Halt*

Unsolvability of the halting problem

- Suppose P solves variant of halting problem
 - On input Q, assume
- $$P(Q) = \begin{cases} \text{yes} & \text{if } Q(Q) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$
- Build program D from P
 - $D(Q) = \begin{cases} \text{run forever} & \text{if } P(Q) = \text{Yes} \\ \text{halt} & \text{if } P(Q) = \text{No} \end{cases}$
 - Does this make sense? What can D(D) do?
 - If D(D) halts, then D(D) runs forever.
 - If D(D) runs forever, then D(D) halts.
 - *CONTRADICTION*: program P must not exist.

- Many “interesting” properties of programs cannot be computed in general
 - Halting Problem
- Frequently used: **Conservative** Algorithms
 - no wrong positive answers, potentially wrong negative answers
- E.g., static type checking is conservative

```
int a,b,c;
...
if (a^3 + b^3 == c^3) {
    int foo = 42 + "bar";
}
```

 - since 1995 we know that this program is type-safe
 - but a type-checker would need to automatically deduce the 100+ page proof of Fermat’s theorem that took 350 years to come up with

Introduction to LISP



- Look at Historical Lisp
 - Invented by John McCarthy in the context of AI
 - Perspective
 - Some old ideas seem old
 - Some old ideas seem new
 - Example of elegant, minimalist language
 - Not C / Java: a chance to think differently
 - Historically one of the most influential languages
 - Based on Lambda calculus
 - “Impure” functional language, i.e., allows assignments
 - Lists are primary data structure (**LISP** = **LIST** Processor)



- Many different dialects
 - Lisp 1.5, Maclisp, ..., Scheme, ...
 - CommonLisp has many additional features
 - This course: a fragment of **Scheme**
- Primitive data types: Integers, Strings, Floats
- Basic Interaction with Scheme: Read-Eval-Print Loop
 - 3 => 3
 - “Foo” => “Foo”
- Function application in prefix notation
 - (+ 5 8) => 13

- Lists are the primary data structure
 - Function application as list
 - (+ 3 2)
 - Lists also basic data structure
 - (1 2 3 4 5)
- How to tell them apart
 - Treat list as data with **quote**
(quote (1 2 3 4 5)) => (1 2 3 4 5)
 - can be abbreviated with **'**
'(1 2 3 4 5) => (1 2 3 4 5)
'(+ 3 4) => (+ 3 4)
- Possible to treat data as program with **eval**
(eval '(+3 4)) => 7

- Functions for lists:
cons car cdr null?

Example

(cons 3 '(4 5)) => (3 4 5)

(car '(3 4 5)) => 3

(cdr '(3 4 5)) => (4 5)

(null? (cdr '(3))) => #t

- Example:

```
(lambda ( parameters ) ( function_body ) )
```

- Syntax comes from lambda calculus:

```
 $\lambda f. \lambda x. f (f x)$ 
```

```
(lambda (f) (lambda (x) (f (f x))))
```

Function expression defines a function but does not give a name to it.

```
((lambda (x) (* 2 x)) 5) => 10
```

Java anonymous inner classes have a similar "flavor"

- Bind top-level names with **define**

```
(define x 5)
```

```
(define y `(1 2 3))
```

```
(define myplus +)
```

- Can also bind name to function

```
(define double (lambda (n) (+ n n)))
```

- Not "pure functional"

– Can replace previous binding

```
(define x 5)
```

```
(define y (lambda () x))
```

```
(y) ;returns 5
```

```
(define x 6)
```

```
(y) ;returns 6
```

Local Variable Bindings

- Bind local names with **let**
(let ((var val) ...) exp) ,e.g.,
(let ((x 5)) (+ x 3) => 8
(let ((+ *)) (+ 2 3)) => 6
- Often used to simplify expression that contains identical subexpressions
(+ (* 4 4) (* 4 4)) => 32
let ((a (* 4 4))) (+ a a) => 32
- **let** is "syntactic sugar" defined in terms of lambda:
(let ((var val)) exp)
is equivalent to
((lambda (var) exp) val)
→ Let is pure functional

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Conditional Expressions in Lisp

- Generalized if-then-else
(cond (p₁ e₁) (p₂ e₂) ... (p_n e_n))
 - Evaluate conditions p₁ ... p_n left to right
 - If p_i is first condition true, then evaluate e_i
 - boolean values: #t and #f
 - Value of e_i is value of expressionUndefined if no p_i true, or
p₁ ... p_i false and p_{i+1} undefined, or
relevant p_i true and e_i undefined
- Special case: only one condition
(if (p e₁) e₂ e₃)

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Examples

(cond ((<2 1) 2) ((<1 2) 1))

has value 1

(cond ((<2 1) 2) ((<3 2) 3))

is undefined

(cond (diverge 1) (#t 0))

is undefined, where diverge is undefined

(cond (#t 0) (diverge 1))

has value 0

Strictness

- An operator or expression form is *strict* if it can have a value only if all operands or subexpressions have a value.
- Lisp cond is not strict, but addition is strict
 - (cond (true 1) (diverge 0))
 - (+ e₁ e₂)

- Function that either
 - takes a function as an argument
 - returns a function as a result
- Example: function composition

```
(define compose
  (lambda (f g) (lambda (x) (f (g x)))))
```
- Example: maplist

```
(define maplist (lambda (f x)
  (if (null? x)
      ()
      (cons (f (car x)) (maplist f (cdr x))))))
```

- What is gained by uniform representation of programs and data?
 - easy to write interpreter in LISP, easy parsing
 - programs as data, e.g.,

```
(write "Which operation do you want")
(define op (read))
(write ( (eval op) 7 8))
```
 - all kinds of source code transformations/generations directly in Scheme!
 - Compare this to Java dynamic class loading!

- “Pure” Functional Programming: no side effects, i.e., no assignments
- No assignments → No sequencing
- Lisp is “impure”, i.e., has assignments and sequencing
 - e.g., Assignment (set! x 3)
 - e.g., Sequencing
(let (var val) ...) exp₁ exp₂ ...
(lambda (var ...) exp₁ exp₂ ...)
- Usage for
 - efficiency, e.g., updating array in quicksort
 - functions with “memory”, e.g., counters, random number generators
- I/O is inherently “impure”

- E.g., counter in LISP

```
(define next 0)
(define counter (lambda () (set! next (+ 1 next)) next))
```
- only one counter, “next” globally visible
- better solution:

```
(define make-counter (lambda ()
  (let ((next 0)) (lambda ()
    (set! next (+ next 1))
    next))))
```

No-side-effects language test

Within the scope of specific declarations of x_1, x_2, \dots, x_n , all occurrences of an expression e containing only variables x_1, x_2, \dots, x_n , must have the same value.

- Example

```
begin
  integer x=3; integer y=4;
  5*(x+y)-3
  ... ||? // no new declaration of x or y //
  4*(x+y)+1
end
```

- Implies: Result of a function depends only on its parameters

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Von Neumann bottleneck

- Von Neumann
 - Mathematician responsible for idea of stored program
- Von Neumann Bottleneck
 - Backus' term for limitation in CPU-memory transfer
- Related to sequentiality of imperative languages
 - Code must be executed in specific order

```
function f(x) { if x<y then y:=x else x:=y };
g( f(i), f(j) );
```
 - Hard to maintain consistency if executed concurrently
 - Deadlocks, Race conditions, ...

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Eliminating VN Bottleneck

- No side effects
 - Evaluate subexpressions independently
 - Example
 - function $f(x)$ { if $x < y$ then 1 else 2 };
 - $g(f(i), f(j), f(k), \dots)$;
 - Concurrency safe **by definition**
- Does this work in practice? Good idea but ...
 - Too much parallelism
 - Little help in allocation of processors to processes
- Effective, easy concurrency is a *hard* problem

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

Reasoning about programs

- To prove a program correct,
 - must consider everything a program depends on
- In functional programs,
 - dependence on any data structure is *explicit*
 - sometimes painfully explicit
- Therefore,
 - easier to reason about functional programs
- Do you believe this?
 - This thesis must be tested in practice
 - Many who prove properties of programs believe this
 - Not many people really prove their code correct
- My opinion: More important whether PL is a good modeling framework for reality
 - i.e., low representational gap
 - the world is full of sequencing and side-effects!

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

- Successful language
 - highly influential on contemporary languages
- Specific language ideas
 - Expression-oriented: functions and recursion instead of sequencing and iteration
 - Lists as basic data structures
 - Programs as data, with universal function `eval`
 - Idea of garbage collection.