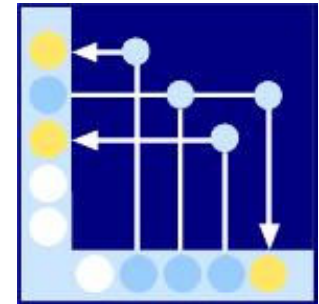




Hochschule Aalen

Fakultät Elektronik und Informatik

Studienbereich Informatik



Advanced Programming with MOSTflexiPL

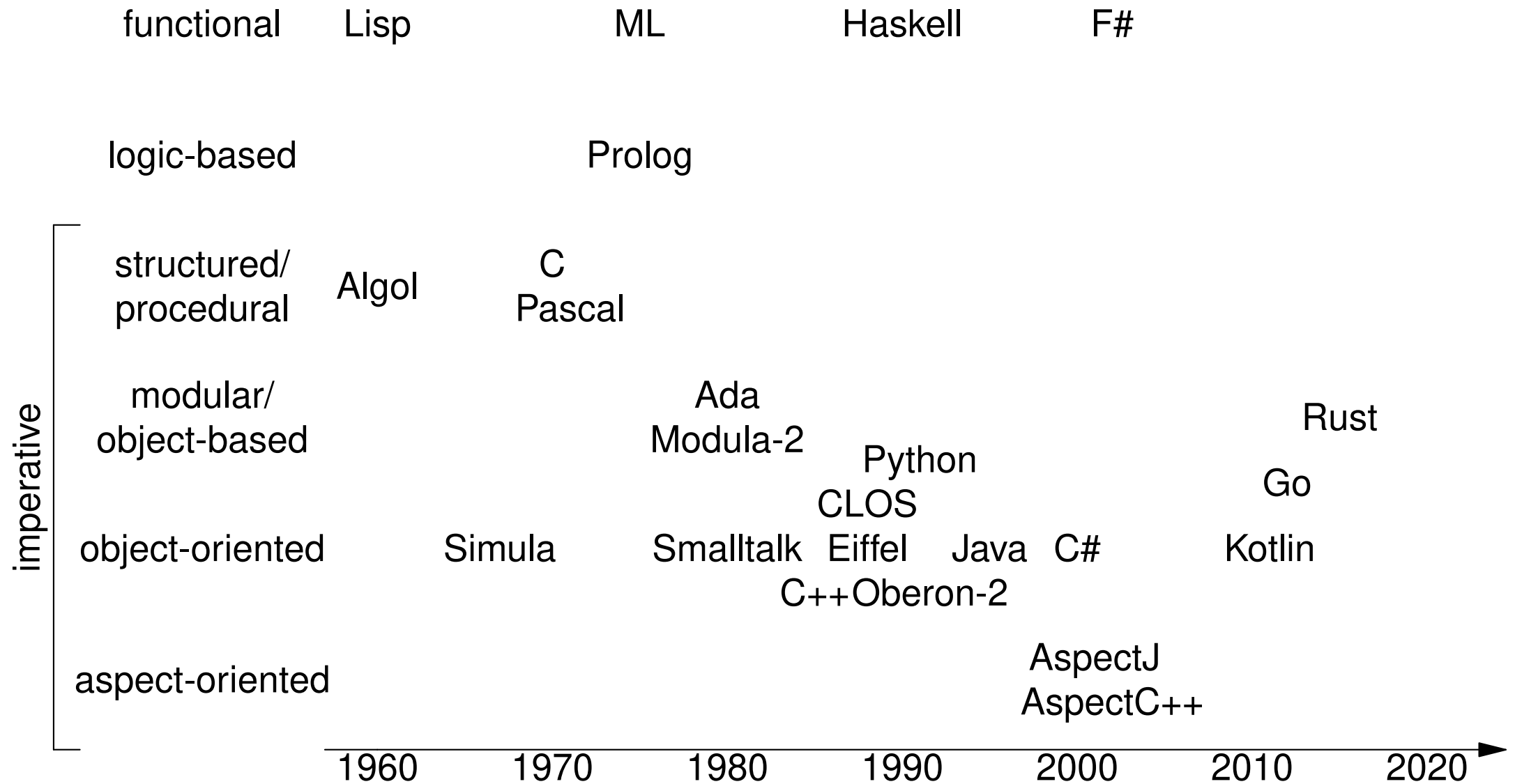
Lecture in Wintersemester 2025/2026

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Introduction

1.1 Well-known Programming Languages and Paradigms



1.2 Imperative and Functional Programming

1.2.1 Difference

- ❑ *Imperative* programs are *state-based*, i. e., they can *modify* the state of their objects by assignments to (global, local, or instance) variables, which are in turn used in conditions of branches and loops and therefore affect a program's behaviour.
- ❑ If an object is used at different locations of a program, a modification at one location can easily have unexpected and undesired side effects at other locations.
- ❑ In contrast, *functional* programs are *state-less*, i. e., there are only constants, but no variables and assignments and therefore no mutable state. By that means, unexpected side effects cannot happen at all.
- ❑ Branches (whose conditions are based on constant values) are still possible, but recursion has to be employed instead of loops.
- ❑ A data structure that is frequently used in functional programs are lists consisting of a first element (head) and a (possibly empty) rest list (tail).
- ❑ Functions can conveniently process such lists by processing the first element directly and the rest list by calling the function recursively.

- ❑ The correctness of such functions can usually be easily proven by induction on the length of the list (mostly with base case 0).

1.2.2 Examples

- ❑ Construction of a list with the natural numbers from m up to n
- ❑ Replacing some value u with some value v in a list of integral numbers

Implementation in the Imperative Programming Language Java

```
List<Integer> nat (int m, int n) {  
    List<Integer> xs = new LinkedList<Integer>();  
    for (int i = m; i <= n; i++) xs.add(i);  
    return xs;  
}
```

```
List<Integer> subst (List<Integer> xs, int u, int v) {  
    List<Integer> ys = new LinkedList<Integer>();  
    for (int x : xs) {  
        if (x == u) x = v;  
        ys.add(x);  
    }  
    return ys;  
}
```

- ❑ The lists `xs` in the `nat` method and `ys` in the `subst` method are repeatedly modified during the loop by calling the `add` method.

Implementation in the Functional Programming Language Haskell

```
nat m n = if m > n then [] else m : (nat (m+1) n)
```

```
subst [] u v = []
```

```
subst (x:xs) u v = (if x == u then v else x) : (subst xs u v)
```

- ❑ Here, every call of the functions `nat` and `subst`, whether directly from outside or via recursion, returns an immutable list of integral numbers.
- ❑ A definition of the form `f x ... = impl` defines a function with name `f`, parameters `x ...`, and implementation `impl`, which can afterwards be called with expressions of the form `f a ...`.
- ❑ An expression of the form `if x then y else z` yields, according to the value of the condition `x`, either the value of `y` or the value of `z`.
- ❑ `[]` denotes the empty list, an expression of the form `x:xs` constructs and returns a new list with first element `x` and rest list `xs`.
(Technically, `x:xs` creates only a new list node containing the element `x` and the pointer `xs` and returns the pointer to this node, which logically represents the whole list.)

- ❑ The two definitions of the `subst` function constitute an implicit branch by so-called *pattern matching*:
If the conveyed list is of the form `[]` (i. e., empty), the first definition is called; if it is of the form `x:xs` (i. e., it contains at least one element `x`), the second definition is called, whose parameters `x` and `xs` will contain the first element and the (possibly empty) rest list of the conveyed list, respectively.
- ❑ As mentioned in § 1.2.1, the correctness of the functions `nat` and `subst` can easily be proven by induction (on the length of the conveyed list in the case of `subst` and on the length $k = \max(n - m + 1, 0)$ of the returned list in the case of `nat`).
- ❑ Even though Haskell is statically typed, the parameter and result types of functions can be omitted, if they can be deduced by the compiler. (By that means, the `nat` function can actually be called for any numeric type, while the `subst` function can be called with lists of any type.)

1.2.3 Remarks

- ❑ Imperative and functional programming are basically just *programming paradigms*, i. e., they define the principal way of doing programming (and the way of thinking while programming).
- ❑ Imperative and functional programming languages provide the mechanisms which are required to do programming in the respective paradigm.
- ❑ Nevertheless, it is usually possible to write functional programs in an imperative programming language (if it provides recursion) by completely abstaining from assignments.
- ❑ However, it is impossible to write imperative programs in a purely functional language due to the absence of variables.

1.3 Meaning of the Name MOSTflexiPL

The acronym MOSTflexiPL (pronounced like *most flexible*, but with *p* instead of *b*) stands for:

☐ **MO**dular

- A comprehensive program can be divided into manageable modules.
- A module can be used as a library in different programs.

☐ **S**tatically **T**yped

- A program is completely checked for being type-correct at compile time.
- Type errors cannot occur at run time.

☐ **f**lexibly **e**xtensible

- The syntax of the language can be easily extended and customized in a virtually unlimited way by every programmer.

☐ **P**rogramming **L**anguage

- It is a general-purpose programming language.

1.4 Descriptive Analogies

1.4.1 Rearrangement of a Room

☐ In a room one can

- ☐ freely move around furniture
- ☐ mount lamps at the ceiling and attach shelves to the walls
- ☐ freely paint or repaper walls
- ☐ and so on

☐ But one cannot simply

- ☐ displace doors and windows
- ☐ move around or remove walls
- ☐ move ceilings up or down
- ☐ and so on

Analogy to Programming Languages

- ❑ With a conventional programming language one can
 - define new data types and functions
 - possibly overload predefined operators (e. g., in C++)
 - possibly define new prefix, infix, and postfix operators (e. g., in Haskell)
- ❑ But one cannot simply
 - extend the syntax of types and declarations
 - define new operators with arbitrary syntax
 - define new control flow statements
- ❑ With MOSTflexiPL, one can do all this and even much more, just as if one could rearrange a room in a completely unlimited way.

1.4.2 Natural Language and Mathematical Notation

❑ In a natural language one may

- define arbitrary new terms and use them afterwards
- define arbitrary new abbreviations and use them afterwards

❑ But one may not simply

- define new punctuation marks and use them afterwards
- change the grammar of the language
- change the rules for capitalization of words
- write text in the opposite direction

❑ In a mathematical article one may

- define arbitrary new symbols with arbitrary notation and use them afterwards,
for example $\sum_{k=1}^n a_k$ or $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$

❑ Therefore, conventional programming languages are limited similarly to natural languages, while MOSTflexiPL offers possibilities comparable to mathematical notation.

1.4.3 Model Railways

❑ A model railway system provides

- straight railtracks of different lengths
- bent railtracks with different radiuses and angles
- different kinds of switches and crossings

❑ But with these things one cannot get

- straight railtracks of any length
- bent railtracks with any radius or angle
- completely different railtrack figures (e. g., ellipses)
- arbitrary switches and crossings
(e. g., switches with more than three ways or crossings with arbitrary angles)

Analogy to Programming Languages

- ❑ Conventional programming languages are limited similarly to normal model railway systems.
- ❑ MOSTflexiPL corresponds to a fictitious model railway system which allows to
 - stretch or squeeze straight railtracks to any desired length
 - shape bent railtracks in an arbitrary way
(this is in fact provided by some manufacturers)
 - build new forms of switches and crossings simply by stacking multiple bent and straight railtracks
- ❑ This flexibility is also expressed by the logo of the language:



1.5 Basic Principles

- ❑ Everything is an *expression*, i. e., the application of an *operator* to subexpressions (*operands*).
- ❑ An operator may have any number of names and operands (depicted by \bullet) in any order, for example (predefined and user-defined operators):
 - Infix operators: addition $\bullet + \bullet$, assignment $\bullet = ! \bullet$
 - Prefix operators: change sign $-\bullet$, logical negation `not` \bullet
 - Postfix operators: faculty $\bullet !$, square \bullet^2
 - Circumfix operators: parentheses (\bullet) , absolute value $|\bullet|$
 - Control flow statements: branch `if` \bullet `then` \bullet `else` \bullet `end`, loop `while` \bullet `do` \bullet `end`
 - Declarations: constant declaration $\bullet : \bullet = \bullet$, operator declaration $\bullet \rightarrow (\bullet = \bullet)$
 - Type expressions: variable type $\bullet ?$, array type $\bullet []$
- ❑ At run time, every expression yields a value.
- ❑ Based on a small set of predefined operators (for arithmetic, logic, control flow), it is possible to define arbitrary user-defined operators.

1.6 Compiler

1.6.1 Availability

- ❑ The MOSTflexiPL compiler `flxc` is installed in the directory `/usr/local/bin` of a virtual bwCloud server running Ubuntu 24.04 LTS. The server's IP address will be announced during the lectures.
- ❑ After a user has been activated by depositing his public SSH key (which can be generated, if necessary, with `ssh-keygen`), he can use SSH with user name `studNNN` (where `NNN` is the user's matriculation number) to log in into this server without a password.
- ❑ During the semester, the compiler will be updated regularly.

1.6.2 Invocation

- ❑ `flxc filename.flx` checks the MOSTflexiPL program in the given source file for syntactic and semantic correctness and creates the corresponding syntax tree.
- ❑ Afterwards, if the program is correct and unambiguous, the syntax tree is directly executed and also saved in binary form in the file `filename.flx.bin`.
- ❑ If the source file (and the compiler) has not been modified since the last successful invocation of the compiler (i. e., if the binary file is newer than the source file and has been produced by the same version of the compiler), the time-consuming correctness check of the program is omitted; in that case, the syntax tree is directly read from the binary file and executed immediately, which is significantly faster for large programs.

1.6.3 Error Messages

- ❑ If the program is not correct, one or more diagnostic messages for possible error reasons are displayed.
- ❑ For that purpose, three main error categories are distinguished:
 - *Insertion errors:*

A subexpression cannot be inserted as an operand into a still incomplete expression, either because its type does not match or because it would violate an exclude specification (cf. § 2.9).

For example: `1 + true`

The subexpression `true` with type `bool` cannot be inserted as an operand into the still incomplete addition `1 +`.
 - *Continuation errors:*

A still incomplete expression cannot be continued, because it is not followed by a matching word in the input.

For example: `if x > 2 else y end`

The still incomplete expression `if x > 2` cannot be continued, because the word `then` which would be necessary for that does not follow in the input.

○ *Completion errors:*

An error is only detected during the final check of an already complete expression. The reasons for such errors are very specific, e. g., that the type of a subexpression cannot be uniquely determined or that an implicit parameter cannot be assigned.

- ❑ The (complete oder incomplete) expressions involved in a possible error are always displayed in the form `12:34 (...) 56:78`, where `12:34` and `56:78` represent the expression's begin and end position, respectively, each of which consists of a line number and a character position within that line, both counted from 1. (The expression starts at the begin position and ends immediately *before* the end position.)
- ❑ The part between the parentheses denotes the expression's operator, e. g., `add/sub: • (+ | -) •` for the predefined operator for addition and subtraction or `true` for the predefined constant `true`.
- ❑ In case of a user-defined operator, the part between the parentheses is itself of the form `21:43 [9] 65:87`, where `21:43` and `65:87` represent the begin and end position, respectively, of the operator's definition, while `9` is the number of the source file containing this definition.
- ❑ The names of the source files corresponding to these numbers are displayed at the end of the whole error output.

- ❑ The readability of the error messages is improved by employing different colors.
- ❑ If `flxc` is called with the option `-v`, even more detailed error messages are produced:
On the one hand, the signature (cf. § 2.8) of user-defined operators is printed in addition; on the other hand, the line(s) of the source file corresponding to each possible error reason are printed in addition.

1.6.4 Display of Ambiguities


- ❑ If the program is ambiguous, one or more diagnostic messages of the following form are displayed:

ambiguity from position 12:34 to 56:78

Here, 12:34 and 56:78 again represent the begin and end position of the ambiguous source code block.


- ❑ In the subsequent lines, this source code block is displayed (without the white space it contains) twice with different colors for each possible interpretation:
- ❑ In the first part of each line (before the vertical bar), each color represents a particular (predefined or user-defined) operator.
The operators corresponding to these colors are shown afterwards also with their respective colors.
- ❑ In the second part of each line (after the vertical bar), each color represents a particular precedence level.
The colors of these precedence levels are red, yellow, green, blue etc. from top (low precedence) to bottom (high precedence) and are shown at the end of the line
ambiguity from ...


Example

ambiguity from position 3:6 to 3:12: operators | precedence levels ()

 | 
 | 

+ operator defined from position 1:1 to 1:20 in module ambig5.flx

+ predefined operator add/sub: 

+ predefined operator 

- ❑ Because the precedence of the operator \cdot^2 (shown in red before the vertical bar) defined in the source file `ambig5.flx` has not been specified, the expression $2 + 3^2$ can be interpreted in two ways which are shown after the vertical bar:
 - Either the square operator (red) is applied to the addition (yellow) of 2 and 3 (both green).
 - Or the addition (red) is applied to 2 (yellow) and the application of the square operator (also yellow) to 3 (green).

1.6.5 Integration into Visual Studio Code

- ❑ An extension for Visual Studio Code, which integrates the compiler into this editor, is provided in the directory `/usr/local/share/flxc` of the virtual server.
- ❑ If this extension is installed in VS Code for Linux, a MOSTflexiPL source file is always checked by the compiler when it is opened or saved (i. e., the code is not checked incrementally). Possible errors or ambiguities are then displayed directly in VS Code.
- ❑ Furthermore, the following features are provided:
 - Simple syntax coloring, e. g., for comments and parentheses.
 - If a source file does not contain any errors, but at most ambiguities, „Go to Definition“ jumps to the definition of a name and „Go to References“ jumps to the locations where a name is used.
- ❑ Whenever the compiler is updated, this VS Code extension is updated correspondingly.

1.6.6 Integration into Vi IMproved

- ❑ A corresponding integration into Vi IMproved is in progress.

1.7 Practical Advice

- ❑ Forget almost everything you are accustomed to from other programming languages!
- ❑ Let your imagination run wild!
Design your “programming room” to your heart’s content!
- ❑ Semicolon is an infix operator. Therefore, it may only appear *between* subexpressions, but not at the end of a „chain“ of expressions.
- ❑ White space between names and operands of an expression is *always* optional.
Therefore, `printlnlyx` instead of `printlnly x` is correct, for example, if there is a suitable constant `x`. (However, `printlnly x` would not be correct: White space is not permitted inside of a single name.)
- ❑ If an error or an ambiguity is not obvious, reduce your program step by step until the problem disappears.
- ❑ On the other hand, develop your programs in many small steps and check every single step by calling the compiler.
- ❑ If an error remains unclear even after extensive investigation, ask me.
(Unfortunately, the compiler is also not completely free of errors.)

1.8 Bibliography

- ❑ C. Heinlein: “MOSTflexiPL – An Extremely Flexible Programming Language.” In: T. Noll, I. Fesefeldt (eds.): *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Aachener Informatik-Berichte AIB-2023-02, Department of Computer Science, RWTH Aachen, 2023, 55–72.

The most recent publication.

- ❑ C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

The most extensive publication yet.

- ❑ C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: S. Jähnichen, B. Rumpe, H. Schlingloff (eds.): *Software Engineering 2012 Workshopband* (Fachtagung des GI-Fachbereichs Software-technik; Februar/März 2012; Berlin). Lecture Notes in Informatics P-199, Gesellschaft für Informatik e. V., Bonn, 2012, 45–60.

A shorter description of the language.

- ❑ C. Heinlein: “Fortgeschrittene Syntaxerweiterungen durch virtuelle Operatoren in MOSTflexiPL.” In: K. Schmid et al. (ed.): *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2014* (Kiel, February 2014). CEUR Workshop Proceedings, 193–212, <http://ceur-ws.org/Vol-1129/paper4A.pdf>.

A description of so-called virtual operators.

All publications except the first use a syntax of the language which is outdated by now.

All of them are available on `flexipl.info/publications`.