A Metalogic-Based Approach to Programming Education

ComputationWorld 2025 and DataSys 2025

April 10, 2025, Valencia, Spain

Hans-Werner Sehring

IARIA

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Outline of the Talk

01 **Programming Education**
How to teach programming – concepts rather than languages

02 **Generalized Compilers**
Compiler technology, programming language and language definitions

03 **Metalogics and M³L**
For describing programming language semantics and syntax

04 **Education Examples**
Examples of teaching programming concepts

05 **Conclusion**
Summary and Outlook

Hans-Werner Sehring

Professor for Software Engineering
Head of the Business Informatics / IT Management (M.Sc.) degree program

Software Engineering

Model-Driven Software Engineering

Evolution-friendly software architecture

Software engineering education

Metamodellierung

Domain Modeling

Software Modeling

M³L

Content Management

Digital communication

Media-based knowledge representation

Personalization

Contact

hans-werner.sehring@**nordakademie**.de
https://www.nordakademie.de/die-hochschule/team/hans-werner-sehring

http://dr.sehring.name

https://**orcid**.org/0009-0008-3016-6868

https://www.**researchgate**.net/profile/Hans-Werner-Sehring

https://scholar.**google**.de/citations?user=hsSrVL8AAAAJ

https://www.**linkedin**.com/in/hwsehring/

Software Engineering

01

Programming Education

The Role of Programming in (CS) Education

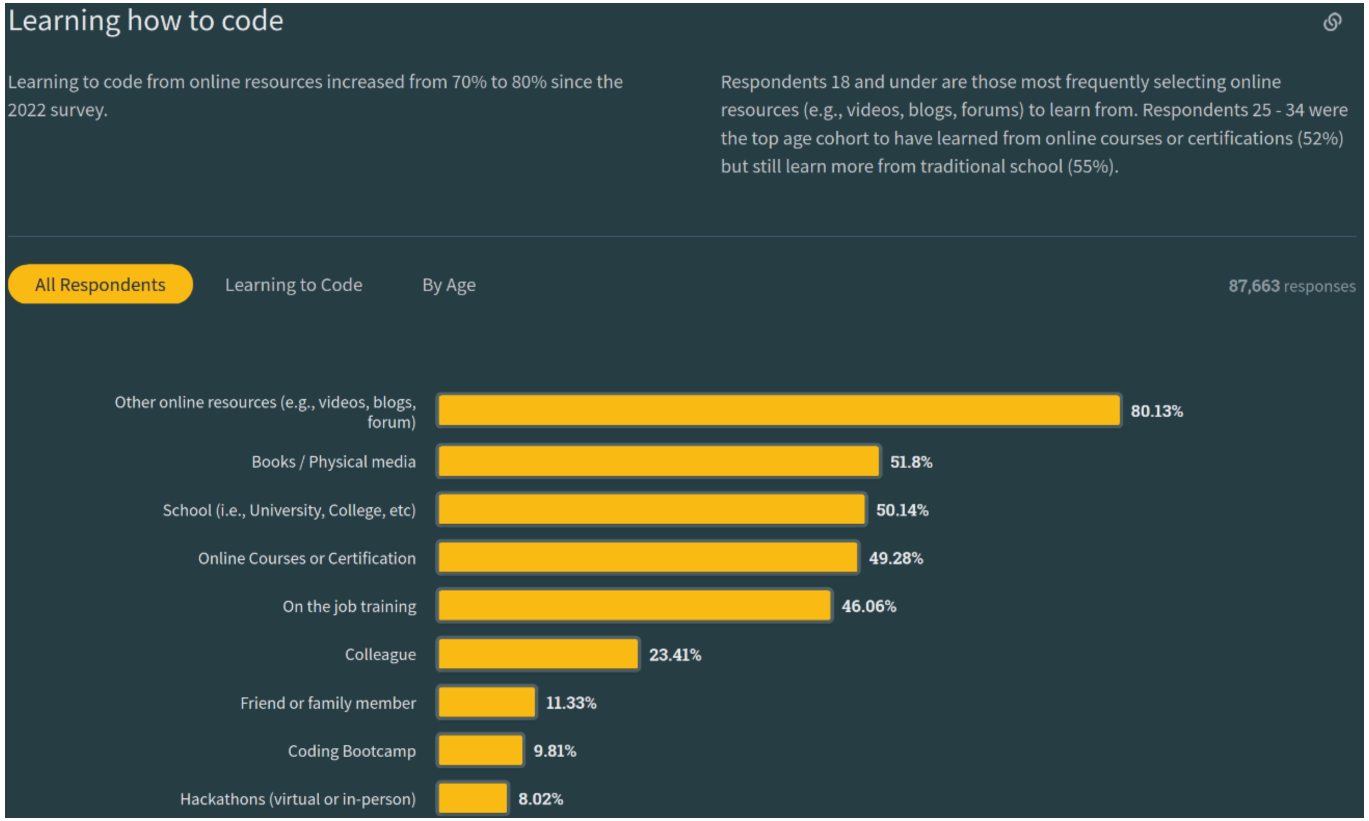
Learning to program is a foundation of each CS curriculum.

For instance,
2023 Developer Survey of Stack Overflow [https://survey.stackoverflow.co/2023/]:

Universities play a central role (still) in programming education – more influential than on the job training

My current context: **NORDAKADEMIE** is part of a dual training, meaning that

- Students are educated with a practical focus
- Education is split between company training and university lecturing



Programming Language Education

Programming is typically taught by the example of a programming language.

Programming Languages (PLs) are still central to computer science (CS)

Programming in general

After decades of (Java, in particular) mono culture, at least in the area of application development, programming became **polyglot** again

Ever new trends change the developers' language preferences, for example,

- Transition from object oriented PLs (OOPLs) to functional PLs (Java development direction, Android, iOS)
- Reactive programming with new demand for concurrent programming

Programming education

Typically, programming is taught using **one or two PLs**

For example, a **scientifically appealing** (functional, in many cases) one and

an **industrially relevant** one (Java, Python, JavaScript, or similar, depending on domain)

Programmers' Wishes are Different

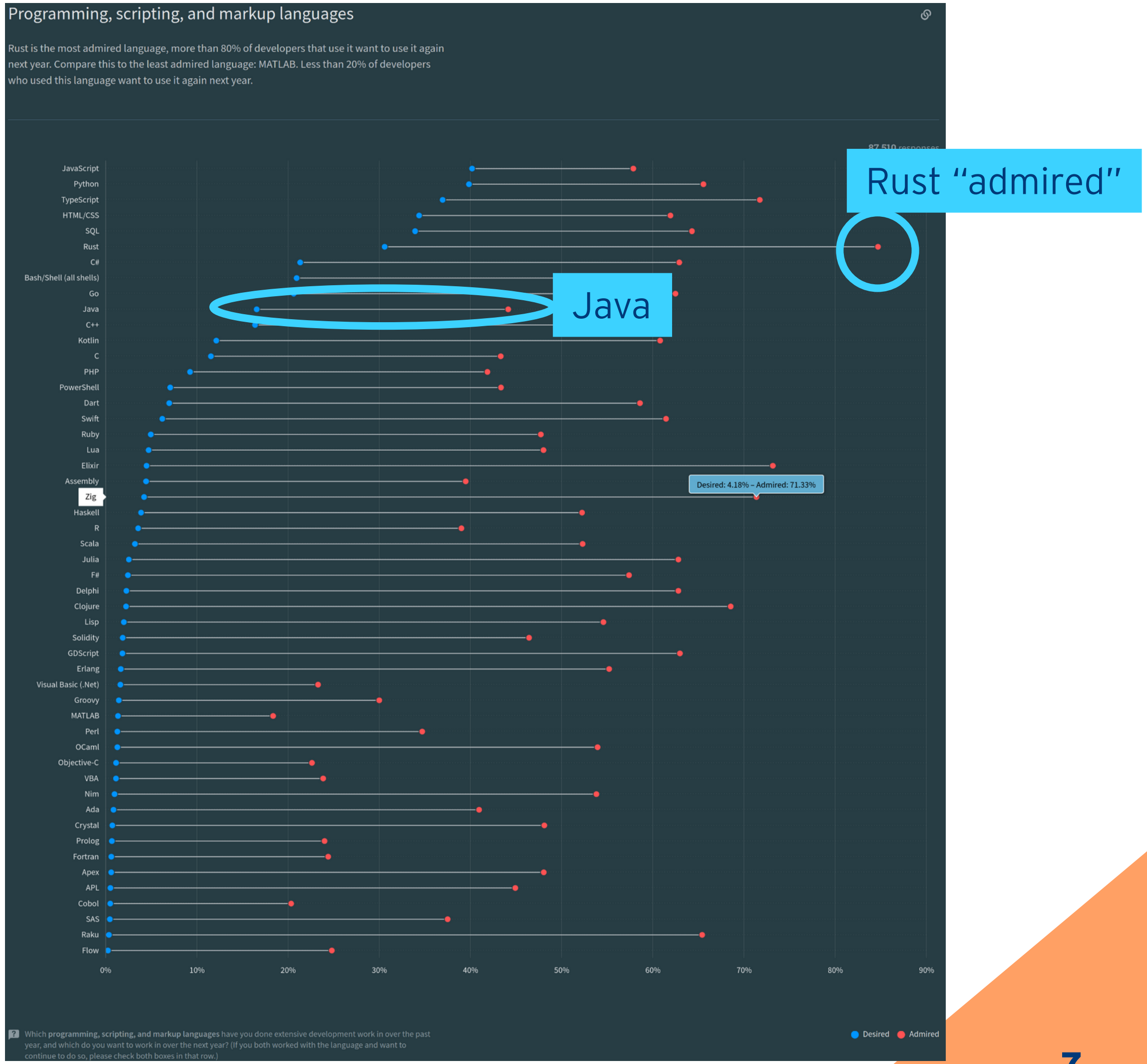
Learning how to program is a foundation of each CS curriculum.

2023 Developer Survey of Stack Overflow [https://survey.stackoverflow.co/2023/]:

"Rust is the most admired language, more than 80% of developers that use it want to use it again next year."

Rust has some nice novel features.

Being targeted at systems programming, Rust will not be most instructors' first choice for beginners' courses
Yet, it is increasingly being used in practice



Programming Education

Changes in programming education might be due.

Rather than teaching languages, it becomes increasingly important (again) to teach **programming concepts**

Reasons for this opinion are manifold

- PLs are **not that long-lived** anymore (again)
- **Polyglot development**: companies choose PLs rather freely, approaches like µService development
- **New forms of "programming"** emerge (for instance, low code, generative AI)
  that free developers from PLs, but not programming techniques
- Some of our students do not program as part of their professional education
  But: programming skills are required in other areas as well: **software architecture**, **process modeling**, etc.

All in all, we should increasingly teach programming concepts instead of concrete programming languages

Programming Paradigms

Students need an overview over programming concepts and their application

Starting point often: **programming paradigms**

Though some are well formalized (functional, lambda calculus),
and others are not (OO, no common object calculus)

Programming paradigms give a first idea of a **mapping of concepts to implementation techniques** and PLs

Example: there are diverse interpretations of object orientation

- Prototype-based vs. class-based
- Class hierarchy with type definitions vs. Mixin (Traits)
- Stateful objects `IntegerSum.new(a,b,c)`
  vs. conversational interfaces `IntegerSum.setSummand(a).setSummand(b).setSummand(c).eval()`
  vs. functional interfaces `a.add(b).add(c)`
- Stateful objects vs. immutable objects
- etc.

Hypothetical Languages for CS Education

Specifically designed languages better allow demonstrating programming concepts.

To teach a range of programming concepts, **many concrete PLs required**

- Different concepts in different PLs
- Existing PLs not paradigm-pure: different interpretations of a paradigm, hybrid languages

Alternative: **hypothetical languages** for particular aspects of programming

Goal:

- Design own hypothetical languages to demonstrate concepts in "pure" form
- According to learning objectives
- Can be done with matured language technology ("yacc"); proposal: language design framework

Example:

OO Java-ish (builtins)	vs.	Smalltalk-ish (Metaclasses)
`class Person`		`Person = ConcreteClass.instanceOf()`
`class Student extends Person`		`Student = Person.subClassOf()`
`mary = new Student`		`mary = Student.new`

Programming is Learned by Practicing

Programming cannot be taught in classroom teaching, it needs hands-on experience.

Programming **needs practice**, both for concrete PLs and for abstract concepts

Additionally, one needs to understand **modeling solutions** with the different approaches

This needs time (per paradigm, ..., concept, ...)

Therefore **new forms of learning** may be due:

- **Flipped learning:** use time to practice and to discuss details
- **Blended learning:** use online resources for learning PL syntax, discuss implications in classroom
- **Active learning:** work with the material, i.e., change it

02

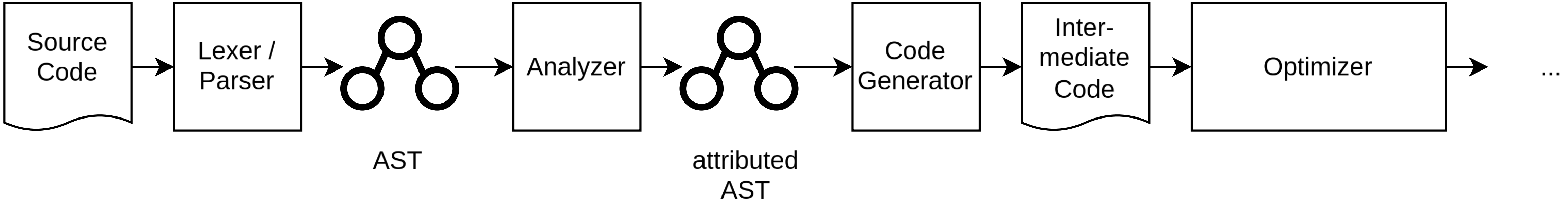
Generalized Compilers

Building Compilers

Building hypothetical programming languages is easy given the existing language tools.

Compiler construction is a well-understood domain including tool support

Typical **compiler architecture**:



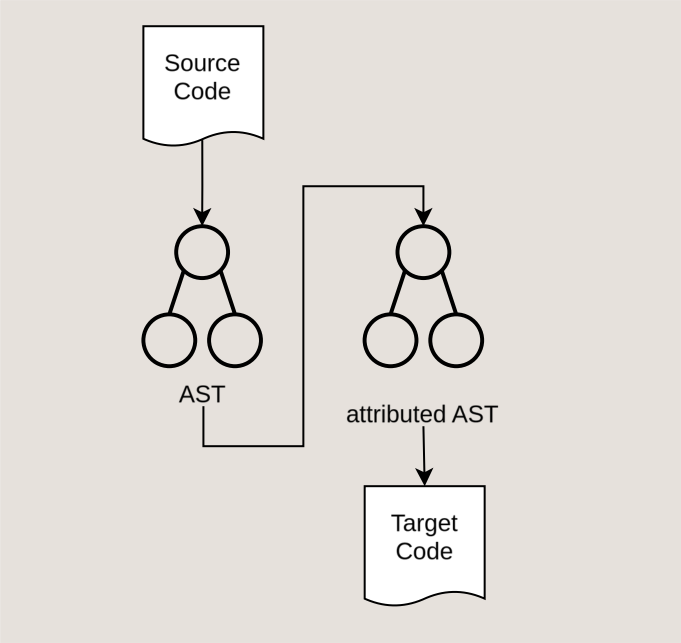
Compiler frontend:

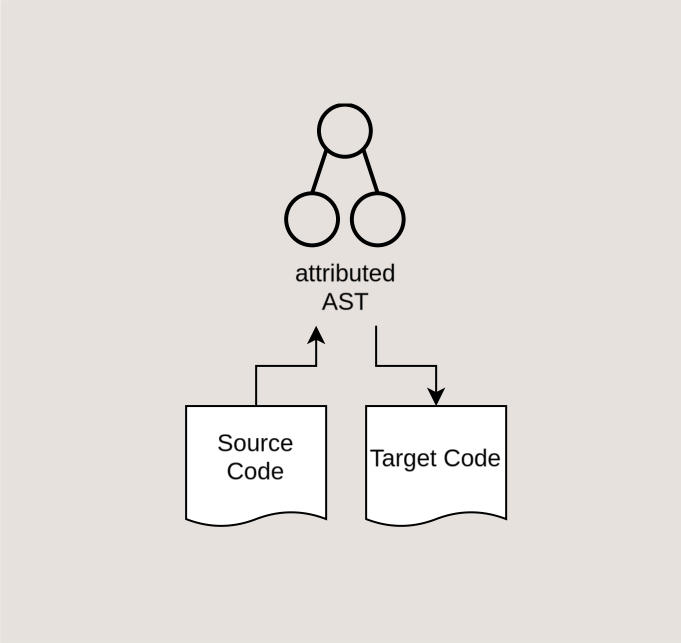
- Source code is tokenized by scanner, producing token stream,
- Structures are recognized by parser, producing **abstract syntax tree** (**AST**),
- Semantic checks are performed by analyzer, resulting in **decorated AST** (links, type information, …)

Then code generation in backend

Traditional Compilers vs. M³L Definitions

Language design with the M³L starts model-driven with a semantically annotated AST.





Compiler Construction

Language **specification** in formal or informal form

Scanner, parser, analyzer etc. **implement** specification

Metalogic-based

Semantics of PL encoded in **model**, concepts and **semantic deduction**

Syntax by **syntactic deduction** by using PL model as a very rich attributed AST

Generalized Compiler

Generalized language tooling based on the idea of the “upside-down” compiler construction

Generalized language tooling based on the idea of the “upside-down” compiler construction

- Design abstract language, including semantics, in the form of a “decorated” AST
- Derive concrete language by adding syntax

For instructors

- In particular for “small” hypothetical PLs used for programming education
- Generalized compiler: create languages from specification
  Requires expressive decoration; new take on PL-defining “decorated” AST: metalogic

For students

- Interact with language specifications: to experience which features are essential, what breaks a PL, etc.
- On top of actual programming: design own languages or modify existing ones

03

Metalogic and M^3L

M³L at a Glance

Basic language constructs. More complete descriptions can be found in the literature.

`A`

The declaration of or reference to a **concept** named A

```
A is a B
A is the B
```

The **refinement** of a concept B to a concept A;
A is a specialization of B, B is a generalization of A (the: A is the only specialization of B)

`A is a B { C }`

Containment of concepts;
C belongs to the **content** of A, A is the **context** of C

`A |= D`

The **semantic rule** of a concept of a concept A;
whenever A is referenced, D is bound;
if D does not exist, it is created in the same context as A

`A |- E F G.`

The **syntactic rule** of a concept A;
A is printed out as or recognized from the concatenation of the syntactic forms of concepts E, F, and G;
if not defined, a concept evaluates to / is recognized from its name

M³L Concept Narrowing and Implicit Subconcepts

Refinement relationships are evaluated when accessing concepts.

```
Person {
  Name is a String }

PersonMary is a Person {
  Mary is the Name }

PersonPeter is a Person {
  Peter is the Name
  42 is the Age }
```

Concepts are analyzed after creation to detect certain constellations

- **Narrowing**

  If a concept **A** has a subconcept **B**, and if all concepts defined in the context of **B** are equally defined in the context of **A**, then each occurrence of **A** is narrowed down to **B**.
  Example: `Person { Peter is the Name 42 is the Age }` is narrowed to `PersonPeter`

- **Implicit Subconcepts**

  If a concept **A** has the same set of base concepts as concept **B**, and if for every content of **A** there is a matching content of **B**, then **A** is a derived base concept of **B**.
  Example: the base concept `PersonMary` is derived for

  ```
  Person { Mary is the Name
           42   is the Age }
  ```

M³L Concept Evaluation

Concept definitions and semantic rules are used to capture concept semantics.

```
Person {
  Name is a String }

PersonMary is a Person {
  Mary is the Name }

PersonPeter is a Person {
  Peter is the Name
  42 is the Age }
```

The M³L has an operational semantics for **concept evaluation**

It is based on (any combinations of)

- Refinement, including implicit refinements
- Semantic rules
- **Visibility** rules
  - All concepts in the content of a concept are also visible in the content of refinements: `A { B }, C is an A` ⇒ `C { B }`
  - All concepts in the content of a concept are also visible in the contents of concepts in the context of that concept: `D E { F }` ⇒ `E { F { D } }`

M³L Concept Representation

Syntactic rules are used to print out and to read in concepts.

```
Person {
  Name is a String
} |- Mr. Name .

PersonMary is a Person {
  Mary is the Name
} |- Mrs. Name .

PersonPeter is a Person {
  Peter is the Name
  42 is the Age }
```

M³L's syntactic rules allow exporting concepts in an external form, and to create / update concepts from such an external form

Such external forms are **formal languages** like programming languages and files formats

Example:

The concept `PersonMary` is externalized as the String `Mrs. Mary`

- The concept `Mrs.` is created when needed
- Both concepts `Mrs.` and `Mary` have no syntactic rule attached

The input `Mr. Smith` leads to the concept `Person { Smith is the Name }` to be created or updated

Programming Paradigms – Imperative PLs

Models of programming paradigms are a good starting point for models of programming.

Type system (any paradigm)

```
Type

Boolean is a Type
True  is a Boolean
False is a Boolean

Integer is a Type {
  Succ is an Integer }
0       is an Integer
PositiveInteger
  is an Integer {
  Pred is an Integer }
1 is a PositiveInteger {
  0 is the Pred }
```

Imperative Basics

```
Statement

Expression
  is a Statement

Variable {
  Name
  Type }

Procedure {
  FormalParameter
       is a Variable
  Body is a Statement }
```

Some Statements

```
ConditionalStatement
  is a Statement {
  Condition is a Boolean
  ThenStatement
    is a Statement
  ElseStatement
    is a Statement }

Loop is a Statement {
  Body is a Statement }

HeadControlledLoop
  is a Loop {
  Condition is a Boolean }
```

PL Semantics

The semantics of the constructs is stated explicitly. There may be alternative realizations.

The semantics of a statement

```
ConditionalStatement
  is a Statement
{
  Condition is a Boolean
  ThenStatement
            is a Statement
  ElseStatement
            is a Statement
}
```

Is given by definitions like

```
IfTrueStmt
 is a ConditionalStatement
{
  True is the Condition
} |= ThenStatement

IfFalseStmt
 is a ConditionalStatement
{
  False is the Condition
} |= ElseStatement
```

Can be used in “programs” like

```
MyConditional
 is a ConditionalStatement
{
  SomePredicate
      is the Condition
  Statement1
      is the ThenStatement
  Statement2
      is the ElseStatement
}
```

`MyConditional` will be a derived subconcept of either `IfTrueStmt` or `IfFalseStmt`

Concrete Programming Languages

Concrete programming languages are implemented by providing syntax rules.

Syntax rules provide a concrete syntax for programming languages, for example

For example, generic OO to Java:

```
Java is an ObjectOrientation {
  ConditionalStatement
  |- if ( Condition )
     ThenStatement
     ElseStatement .
}
```

Generic OO to Python:

```
Python is an ObjectOrientation {
  ConditionalStatement
  |- if Condition :
     "  " ThenStatement
     else:
     "  " ElseStatement .
}
```

Typically, there is no direct mapping of general concepts to PLs

- Languages implement concepts differently. For example, Java misses some object-oriented features and expresses them differently. Intermediate models bridge the gap between programming models in “pure form” and concrete PLs
- Many languages are hybrid in nature, so that more than multiple programming model are combined

04

Education Examples

PL Education

Certain properties of programming languages are central to understanding programming.

There are various examples of **basic PL education** that are hard to understand for beginners

- persistent data of functional PLs (and mutation will break many of them)
- mutable data of imperative PLs (and all associated problems)
- parameter passing by value, by reference, and by name
- scopes and contexts (scope in structured programming, objects, closures, etc.)
- the theory of OO type systems (subtyping, inheritance, variance, `Null` singleton, `Void` singleton, etc.)
- everything related to concurrent programming

Often, **few of them are covered in sufficient detail** (depending on the teaching approach), while others are touched remotely

So far, we use dedicated PLs to discuss some features
Experiments to demonstrate them using the M3L are currently on the level of logic, not suitable for students

Example 1: Understanding Functional Programming

Students need to understand why functional programming uses immutable data.

Example 1: persistent data in functional programming

Assuming a base definition of programming concepts in a context `FunctionalProgramming`, the following might be asked

```
> MyFunProg is a FunctionalProgramming { i is an Identifier }
MyFunProg
> MyFunProg { iDecl1 is a Declaration { i is the DeclaredIdentifier
                                        1 is the Value } }
MyFunProg
> MyFunProg { iDecl1 is a Declaration { i is the DeclaredIdentifier
                                        2 is the Value } }
Error: "i" has been defined to be the only base concept of "1"; cannot
assign further base concepts
```

M3L error message not helpful;
should report something like:
"identifier i already defined with value 1, cannot be assigned another value"

Example 2: Understanding Imperative Programming

Students need to understand scopes to master imperative programming.

Example 2: variable scopes in imperative programming

Assuming a base definition of programming concepts in a context `ImperativeProgramming`, the following might be asked

```
> MyImpProg is an ImperativeProgramming {
  i is a Variable { Integer is the Type }
  1 is the i
  MainProgram is a Procedure {
    i is a Variable { Integer is the Type }
    2 is the i
  } is the MainProcedure
}
MyImpProg
> i from MyImpProg
1
> i from MainProgram from MyImpProg
2
```

05

Conclusion

Conclusion

Summary and Outlook

Summary

They way programs are constructed changes in some areas

Yet, a proper education in basic programming techniques is required

To account for the various aspects of programming, either a lot of PLs and other tools have to be used in teaching, or a universal

The latter is one possible research objective

Outlook

A first step will be identifying minimal versions of actual PLs or hypothetical PLs that exhibit the features to be taught

The potential of the environment that the M^3L provides shall be researched

Only with concrete PLs it will be possible to provider better error messages (in terms of the chosen PLs) etc.

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

NORDAKADEMIE gAG Hochschule der Wirtschaft
Köllner Chaussee 11 · 25337 Elmshorn · Tel.: +49 (0) 4121 4090-0 · E-Mail: info@nordakademie.de · Web: www.nordakademie.de