Programming Languages in Software Engineering

Lecture 1

Komi Golova (she/her)
komi.golov@jetbrains.com

Constructor University Bremen

About the course

Premise

This course is a historical oddity.

Idea: One master, three tracks. Everyone should see a bit of all three.

Premise

This course is a historical oddity.

Idea: One master, three tracks. Everyone should see a bit of all three.

Reality: 9 ML students.

Premise

This course is a historical oddity.

Idea: One master, three tracks. Everyone should see a bit of all three.
Reality: 9 ML students.

How do we make a relevant, useful, and fun course from this?

Considerations

Our goals:

- Teach you the basics of PL work
- Broaden your horizons

Considerations

Our goals:

- Teach you the basics of PL work
- Broaden your horizons

Our context:

- Programming is (maybe) changing
- We want to give you skills for the future
- We want you to shape that future

Considerations

Our goals:

- Teach you the basics of PL work
- Broaden your horizons

Our context:

- Programming is (maybe) changing
- We want to give you skills for the future
- We want you to shape that future

My background: master's degree in maths, some years of industry experience, half a PhD in maths and CS.

Considerations

Our goals:

- Teach you the basics of PL work
- Broaden your horizons

Our context:

- Programming is (maybe) changing
- We want to give you skills for the future
- We want you to shape that future

My background: master's degree in maths, some years of industry experience, half a PhD in maths and CS.

Your experience: machine learning.

Course vision

We focus on a question:

"What does the language of the future look like?"

My contribution:

- How do we build a language?
- What languages (can) exist?
- How are ML and PL combined?

Your contribution:

- What could we do better?
- How do we train a model?
- How do we measure our results?

Hopefully, together we can build something cool.

Course structure

- Weekly lectures
  - PL theory
  - Case studies
  - (Hopefully) guest lectures
- (Mostly) team-based homework assignments
  - Relevant to the goal of building a language
- Instead of seminars: 30-minute meetings between me and each team
- Grading:
  - 50% homework
  - 25% final presentation
  - 25% meetings

First homework will be communicated via Telegram, due 21/9.

AI: General

Discussion

What is AI good at?

What is AI bad at?

Varieties of programming languages

“Normal programming”

What features do we expect?

“Normal programming”

What features do we expect?

- Lexical scope
- Nested scope
- Eager evaluation
- Sequential evaluation
- Mutable first-order variables
- Mutable first-class structures
- Higher-order functions
- Closures
- Automated memory manangement

— Identifying and Correcting Programming Language Behavior Misconceptions, Lu and Krishnamurthi, OOPSLA, 2024

Functional

Main feature: focus on values

```
val ys = xs.filter { x > 0 }
           .mapIndexed {
              i, x -> i * x
            }
           .take(20)
```

Common features:

- Higher order functions
- Closures
- Controlled effects

Examples:

- Haskell family:
  - Haskell, Agda, Idris
  - Monads for effects
- ML family:
  - SML, OCaml, F#
  - Awesome modules
- LISP family:
  - Common LISP, Racket, Clojure, Scheme
  - Homoiconic syntax

Logical

Main feature: search.

```
add(X, Y+1, Z+1) :-
    add(X, Y, Z).
add(X, 0, X).
```

Common features:

- Parameters are both inputs and outputs.
- Order of evaluation unspecified.

Examples:

- Prolog
  - Historic connections to AI
- Erlang
  - Used for web applications
- MiniKanren:
  - Minimalistic language
- Verse
  - Made by EpicGames
  - Used for Fortnight
  - Multi-value expressions

Actor-based

Main feature: communication through messages

```
actor UserStorage {
  private var users = ...
  func store(...) { ... }
}
```

Common features:

- Safe concurrency

Examples:

- Objective-C, Swift
  - Since Swift 5.5, built into the language
- Erlang

Fun fact: this is is the “original” meaning of object-oriented.

Dependently typed

Main feature: no distinction between types and terms

```
def reverse {n} {α} : Vec n α
                    → Vec n α
```

Common Features:

- Totality
- Functional style
- Support for verification
- Interest in equality

Examples:

- Rocq, Lean
  - Tactic-oriented
- Agda, Arend
  - Homotopy type theory
- Idris
  - Linear logic
- F*
  - Built-in SMT solver

Array

Main feature: automatic dimension polymorphism

```
2 * pd.Series(1, 2, 3)
```

Common features:

- Variations on the above
- Conciseness

Examples:

- APL, J, K
  - Arcane magic
- Uiua
  - Modern arcane magic
- NumPy, Pandas, MATLAB
  - Focus on data science

Concatenative

Main feature: values passed implicitly

```
input:
 ◿5 ↯3_4 ⇡12
```

```
output:
╭─
╷ 0 1 2 3
  4 0 1 2
  3 4 0 1
          ╯
```

Common features:

- Extreme conciseness
- Array-based

- APL, J, K
  - Arcane magic
- Uiua
  - Modern arcane magic
- FORTH
  - Minimalistic concatenative language

And more!

- Reactive: computation is in response to things changing.
- Metaprogramming: programs are generated by programs
- Declarative: programs describe what should happen
  - Very broad category, but e.g. SQL has no better one

What others do you know?

AI: Features

Discussion

What programming language features impact AI performance?

Intro to PL

What is PL?

What do we do when we study calculus?

What is PL?

What do we do when we study calculus?

We look at the objects that exist (functions $\mathbb{R} \to \mathbb{R}$) and look at how they behave.

What is PL?

What do we do when we study calculus?

We look at the objects that exist (functions $\mathbb{R} \to \mathbb{R}$) and look at how they behave.

PL is different: largely we do not *have* objects that exist.
Instead: we generate the objects to talk about.

What is PL?

What do we do when we study calculus?

We look at the objects that exist (functions $\mathbb{R} \to \mathbb{R}$) and look at how they behave.

PL is different: largely we do not *have* objects that exist.
Instead: we generate the objects to talk about.

We can generate anything, but usually we go for trees.
Once we've made things, we need to show they behave well.

Trees

Trees are nice to generate, because they grow recursively from a root.

Examples: grammars, data types, proof trees...

```
SUM -> PROD '+' SUM | PROD
PROD -> ATOM '*' PROD | ATOM
ATOM -> NUM | '(' SUM ')'
```

```
data Bin a = Tree (Bin a) (Bin a) | Leaf a
```

$$\frac{}{\mathrm{LE}(0, n)}\,\text{zero} \qquad \frac{\mathrm{LE}(k, n)}{\mathrm{LE}(k+1, n+1)}\,\text{successor}$$

When you see "inductively defined", think "tree-shaped".

What does PL research look like?

Let's inductively define:

- An abstract syntax
  - What programs are well-formed?
- A type system
  - What programs are valid?
- An operational semantics
  - How do programs compute?

What does PL research look like?

Let's inductively define:

- An abstract syntax
  - What programs are well-formed?
- A type system
  - What programs are valid?
- An operational semantics
  - How do programs compute?

This is all "free". But we want real properties, like "well-typed programs do not get stuck", so then we have to start proving things.

"For every program (tree) if it has a type (tree, witnessed by a tree) then every reduction sequence (witnessed by a tree) is not stuck."

What does PL implementation look like?

Let’s inductively define an abstract syntax.

(That is: specify what shapes our trees may have.)

What does PL implementation look like?

Let's inductively define an abstract syntax.
(That is: specify what shapes our trees may have.)

What does our implementation do?

1. Start with a string
2. Quick, turn it into a tree (parsing)
3. Okay, let's do tree stuff to it:
   - Check it's fine (type-checking, static analysis)
   - Run it (interpretation)
   - Turn it into something else (compilation)

What does PL implementation look like?

Let’s inductively define an abstract syntax.
(That is: specify what shapes our trees may have.)

What does our implementation do?

1. Start with a string
2. Quick, turn it into a tree (parsing)
3. Okay, let’s do tree stuff to it:
   - Check it’s fine (type-checking, static analysis)
   - Run it (interpretation)
   - Turn it into something else (compilation)

When you walk a tree, the path to the root is a stack.

AI: Methods

Discussion

How do we measure AI performance?

How can we do this across languages?

How can we do this cheap?