# Programming Languages in Software Engineering

Lecture 4

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

Constructor University Bremen

## Project outline

Two teams:

- LLM-friendly language (Nikita Brescanu)
  - Key question: what syntax works best for LLMs?
- LLM-integrated language (Anton Korotkov)
  - Key question: what language features do LLMs enable?

Timeline:

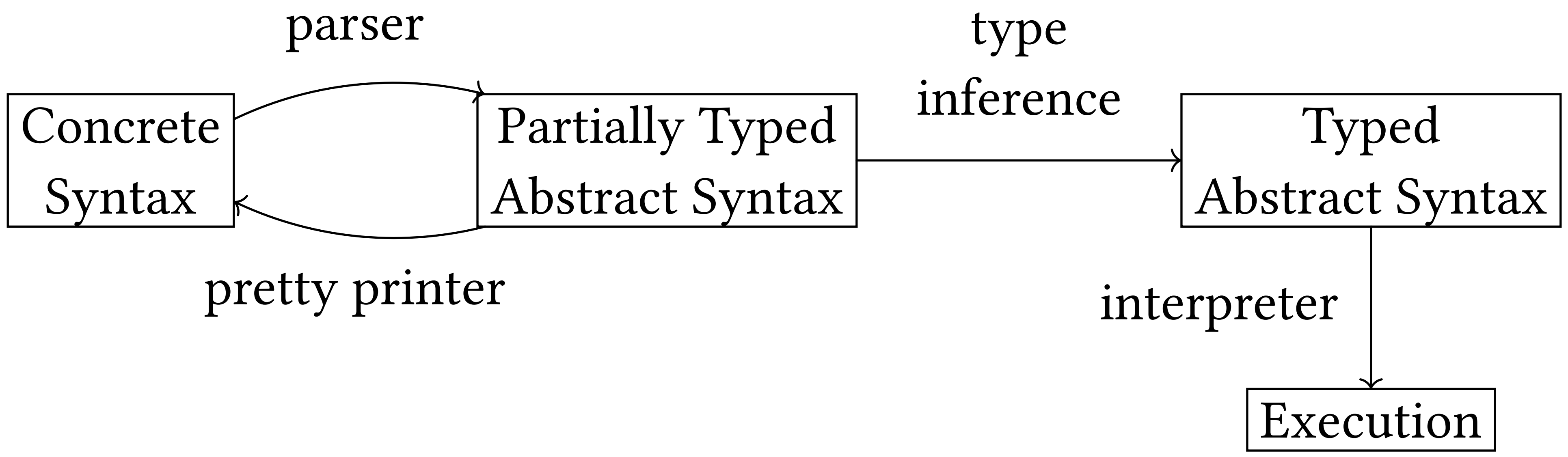
- 21/10: implement basic language
- 4/11: specify distinguishing feature and evaluation criteria
- 18/11: implement distinguishing feature
- 2/11: evaluate distinguishing feature and present results

## Basic language

We will focus on a language suitable for data science. Key features:

- Array-oriented programming
- Static types

Overview of an implementation:



## Example

Concrete syntax:

```
fn inc(x: [Int]): [Int] { x+1 }
```

## Example

Concrete syntax:

```
fn inc(x: [Int]): [Int] { x+1 }
```

Partially typed abstract syntax:

```
FunDef("inc", [("x", ArrayOf(IntType))], ArrayOf(IntType),
  OpCall(PlusOp, [Var("x"), Literal(1)]))
```

## Example

Concrete syntax:

```
fn inc(x: [Int]): [Int] { x+1 }
```

Partially typed abstract syntax:

```
FunDef("inc", [("x", ArrayOf(IntType))], ArrayOf(IntType),
  OpCall(PlusOp, [Var("x"), Literal(1)]))
```

Typed abstract syntax:

```
FunDef("inc", [("x", ArrayOf(IntType))], ArrayOf(IntType)
  OpCall(PlusOp,
         [Var("x", ArrayOf(IntType)), Literal(1, IntType)],
         ArrayOf(IntType)))
```

## Distinguishing features

To make the language a better fit for data science, let’s make everything shape-polymorphic:

```
const xs = [1, 2, 3]
1 + xs // [2, 3, 4]
xs + xs // [2, 4, 6]
Int square(Int n) = n*n
square(xs) // [1, 4, 9]
```

## Assignment overview

In this lecture, I will explain the basic AST structure that your language must support.

## Assignment overview

In this lecture, I will explain the basic AST structure that your language must support.

You will need to design a concrete syntax for this abstract syntax and implement:

- A parser
- A pretty printer
- A type inference engine
- An interpreter
- A standard library
- A test suite for the above

You may use any means, including any kind of AI, to get this done.

## Abstract vs concrete syntax

The same abstract syntax can be represented in many ways.

```
FunDef("inc", [("x", ArrayOf(IntType))], ArrayOf(IntType),
  OpCall(PlusOp, [Var("x"), Literal(1)]))
```

## Abstract vs concrete syntax

The same abstract syntax can be represented in many ways.

```
FunDef("inc", [("x", ArrayOf(IntType))], ArrayOf(IntType),
  OpCall(PlusOp, [Var("x"), Literal(1)]))
```

Examples:

```
fn inc(x: [Int]): [Int] { x+1 }
Int[] inc(Int x[]) { return x+1; }
(defun ((f (Array Int)) (x (Array Int))) (+ x 1))
decl inc Int [] Int [] x func x 1 +
inc:xI^→I^.x1+
DECLARE FUNCTION inc
  TAKING ARRAY OF INTS x
  RETURNING ARRAY OF INTS ...
```

## Tree structure

We can classify the nodes of our AST as follows:

- `Program`: root, list of declarations
- `Declaration`: introduction of an entity (e.g. function)
- `Statement`: step of execution (e.g. assignment)
- `Expression`: fragment that evaluates to a value (e.g. literal)
  - `PlaceExpression`: an expression that denotes a location
- `Type`: classifier of expressions (e.g. record name)

There may be different concrete nodes for the above.
Each concrete node is represented by a `dataclass`.

Your code should be able to work with structures as defined below, but you may add more nodes or add extra data to nodes.

## Partially Typed AST

## Top-level structure

Every program is a sequence of declarations.

```
class Program:
    declarations: list[Declaration]
```

These declarations can be:

- Global variables
- Functions
- Record types

Program execution starts by executing the `main` function.

## Variable declarations

Variables have a name, type, mutability, and optional initializer.

```
class VarDecl:
    name: str
    type: Type
    mutable: bool
    initializer: Optional[Expression] = None
```

## Variable declarations

Variables have a name, type, mutability, and optional initializer.

```
class VarDecl:
    name: str
    type: Type
    mutable: bool
    initializer: Optional[Expression] = None
```

Note: `mutable` does not dictate how we specify mutability. Examples:

- `let x: i32` vs `let mut x: i32` vs `let x: mut i32`
- `int x;` vs `const int x;`
- `var x: Int` vs `val x: Int`

## Function definitions

```
class FunctionDef:
    name: str
    params: list[VarDecl]
    return_type: Type
    body: Expression
```

## Function definitions

```
class FunctionDef:
    name: str
    params: list[VarDecl]
    return_type: Type
    body: Expression
```

Why have function declarations rather than treat functions as variables?

## Function definitions

```
class FunctionDef:
    name: str
    params: list[VarDecl]
    return_type: Type
    body: Expression
```

Why have function declarations rather than treat functions as variables? Inference works differently: we tend to infer closure return types, but want to have function return types available.

## Function definitions

```
class FunctionDef:
    name: str
    params: list[VarDecl]
    return_type: Type
    body: Expression
```

Why have function declarations rather than treat functions as variables? Inference works differently: we tend to infer closure return types, but want to have function return types available.

Note: the body is an `Expression`, not a `Statement`; the expression's result is the return value.

## Record type declarations

Records are the equivalent of Python's dataclasses.

```
class RecordTypeDecl:
    name: str
    fields: list[VarDecl]
```

You may generalise `VarDecl` to `Declaration` here to add support for member functions.

## Declarations

Putting the above together, `Declaration` is the union of the different kinds of declarations:

```
Declaration = FunctionDef | VarDecl | RecordTypeDecl
```

You can pattern match on declarations like this:

```
match decl:
    case FunctionDef(name, params, ret, body):
        # handle function
    case VarDecl(name, ty, mut, init):
        # handle variable
    case RecordTypeDecl(name, fields):
        # handle record type
```

## Statements and expressions

A statement is a sequential block of computation.

An expression is a fragment that evaluates to a value.

## Statements and expressions

A statement is a sequential block of computation.
An expression is a fragment that evaluates to a value.

As programming has become more value-oriented, more and more things that used to be statements are now expressions:

```
if (x) {
  y = a;
} else {
  y = b;
}
```

```
y = if (x) { a } else { b }
```

(Yes, C has `x ? a : b`, but it is much more niche.)

## Statements and expressions

A statement is a sequential block of computation.
An expression is a fragment that evaluates to a value.

As programming has become more value-oriented, more and more things that used to be statements are now expressions:

```
if (x) {
  y = a;
} else {
  y = b;
}
```

```
y = if (x) { a } else { b }
```

(Yes, C has `x ? a : b`, but it is much more niche.)

Our language follows this modern trend: most things are expressions.

## Expressions

Expressions are evaluated for their value.

## Expressions

Expressions are evaluated for their value.

Positive: easy to build complex expressions.
Negative: order of side effects can be hard to reason about.

Example: `++i + i++` in C or C++ has undefined behaviour.

## Expressions

Expressions are evaluated for their value.

Positive: easy to build complex expressions.
Negative: order of side effects can be hard to reason about.

Example: `++i + i++` in C or C++ has undefined behaviour.

Compromise:

- We do allow some side effects in expressions (e.g. function calls).
- We don't allow assignments in expressions, except…
- Blocks are expressions, and blocks may contain statements (including assignment statements).

## Expressions

Expressions are evaluated for their value.

Positive: easy to build complex expressions.
Negative: order of side effects can be hard to reason about.

Example: `++i + i++` in C or C++ has undefined behaviour.

Compromise:

- We do allow some side effects in expressions (e.g. function calls).
- We don't allow assignments in expressions, except…
- Blocks are expressions, and blocks may contain statements (including assignment statements).

Forbidden: `(x = 5) + x`

Allowed: `{ x = 5; x } + x`

## Literals

The simplest expressions are literal values, usually called “literals”.

```
class PrimitiveLiteral:
    value: int | float | bool
class LambdaLiteral:
    params: list[VarDecl]
    body: Expression
```

```
class ArrayLiteral:
    value: list[Expression]
class RecordLiteral:
    type: str
    field_values:
      dict[str, Expression]
```

## Literals

The simplest expressions are literal values, usually called "literals".

```
class PrimitiveLiteral:
    value: int | float | bool
class LambdaLiteral:
    params: list[VarDecl]
    body: Expression
```

```
class ArrayLiteral:
    value: list[Expression]
class RecordLiteral:
    type: str
    field_values:
      dict[str, Expression]
```

The right column here is a little bit of a lie:

- Constructs on the left directly correspond to a single value.
- Constructs on the right may have unevaluated subexpressions.

It is thus a slight abuse of the term "literal value".

## Function and operator calls

Function and operator calls work similarly. We split them out to make it easier to pretty-print operator calls differently.

```
class FunctionCall:
    function: Expression
    arguments: list[Expression]
class OperatorCall:
    operator: str
    operands: list[Expression]
```

## Function and operator calls

Function and operator calls work similarly. We split them out to make it easier to pretty-print operator calls differently.

```
class FunctionCall:
    function: Expression
    arguments: list[Expression]
class OperatorCall:
    operator: str
    operands: list[Expression]
```

Notes:

- The `function` can be any expression.
- Evaluation order: function, then arguments/operands left-to-right.

## Function and operator calls

Function and operator calls work similarly. We split them out to make it easier to pretty-print operator calls differently.

```
class FunctionCall:
    function: Expression
    arguments: list[Expression]
class OperatorCall:
    operator: str
    operands: list[Expression]
```

Notes:

- The `function` can be any expression.
- Evaluation order: function, then arguments/operands left-to-right.

Language feature: all functions and operators are shape-polymorphic. We will go into what this means later.

## If expressions

```
class IfExpr:
    condition: Expression
    then_expr: Expression
    else_expr: Expression
```

Type requirements:

- `condition` must evaluate to a `Bool`
- `then_expr` and `else_expr` must evaluate to the same type.

## Block expressions

Block expressions allow you to use statements.

```
class Block:
    statements: list[Statement]
```

## Block expressions

Block expressions allow you to use statements.

```
class Block:
    statements: list[Statement]
```

What is the result?

- If the last statement is an expression: its result.
- If the last statement is not an expression: unit.
  - Useful for function bodies.
- If the list is empty: also unit.

## Block expressions

Block expressions allow you to use statements.

```
class Block:
    statements: list[Statement]
```

What is the result?

- If the last statement is an expression: its result.
- If the last statement is not an expression: unit.
    - Useful for function bodies.
- If the list is empty: also unit.

The unit value is the only value of the `Unit` type.

Older programming languages call this type `Void`.

## Place expressions

In general, expressions denote values, but some also denote places.

```
class VarRef:
    name: str
```

```
class FieldRef:
    record: Expression
    field_name: str
```

These are expressions that can appear on the left-hand side of an assignment statement.

## Statements

A statement is a piece of code that comes in a clear order with respect to the surrounding code.

We have the following statements:

- Assignments
- While loops
- Declaration statements
- Expression statements

We don't have return statements; instead, the last expression is returned.

## Assignment statement

```
class Assignment:
    lvalue: PlaceExpression
    rvalue: Expression
```

## Assignment statement

```
class Assignment:
    lvalue: PlaceExpression
    rvalue: Expression
```

Note that evaluation order can still be worth thinking about:

`f().x = g()`: which is called first, `f` or `g`?

## Assignment statement

```
class Assignment:
    lvalue: PlaceExpression
    rvalue: Expression
```

Note that evaluation order can still be worth thinking about:

`f().x = g()`: which is called first, `f` or `g`?

Convention: fully resolve `lvalue`, then `rvalue`, then assign.

## While loops

```
class WhileLoop:
    condition: Expression
    body: Statement
```

Note that the body is a statement, unlike with if expressions.

Usually, it doesn't make sense to return a value from a while loop.

## While loops

```
class WhileLoop:
    condition: Expression
    body: Statement
```

Note that the body is a statement, unlike with if expressions.
Usually, it doesn't make sense to return a value from a while loop.

We don't have for loops to minimize the required work—feel free to add them.

## Declaration and expression statements

Declarations and expressions can also be used as statements.

```
class DeclStmt:
    declaration: Declaration


class ExprStmt:
    expression: Expression
```

However: you do not need to support local record type declarations.

## Quick recap

We introduced the executable part of the syntax:

- How to define functions and variables.
- How to compute with values.
- How to do things sequentially.

We are missing a syntax for types.

## Types

Continuing our theme of array-orientation, we will make arrays a fundamental part of our type system.

We identify three kinds of base types:

- Primitives
- Functions
- Records

## Types

Continuing our theme of array-orientation, we will make arrays a fundamental part of our type system.

We identify three kinds of base types:

- Primitives
- Functions
- Records

To form a `Type` from a `BaseType`, we annotate it with its dimension. For scalars, this is 0, while for arrays it is positive.

## Types

Continuing our theme of array-orientation, we will make arrays a fundamental part of our type system.

We identify three kinds of base types:

- Primitives
- Functions
- Records

To form a `Type` from a `BaseType`, we annotate it with its dimension. For scalars, this is 0, while for arrays it is positive.

Note: In this section, we talk about the *syntax* of types.
It doesn't matter what they mean, just what we can write.

## Primitive types

```python
class PrimitiveType:
    name: str
```

Specifically, you must support the following primitive types:

- `Unit`
  - The Python equivalent of this is `None`
- `Int`
- `Boolean`
- `Float`

## Record types

Our system is nominally typed:

- `record Coord { x: Float, y: Float }` declares a type.
- However, `{ x: Float, y: Float }` is *not* a type.

Hence a record type is simply given by its name:

```
class RecordType:
    name: str
```

## Function types

A function type is given by its parameters and return type.

```
class FunctionType:
    param_types: list[Type]
    return_type: Type
```

## Array types

As stated above, primitives, records, and functions are only base types.

Full types must have a dimension:

```
class Type:
    base_type: BaseType
    dimension: int
```

This representation lets us be more consistent with dimension annotations, and avoid mismatch issues we could otherwise have like

`Array[n](Array[m](Int))` vs `Array[m](Array[n](Int))`

## Type system notes

Syntactically, we have a quite simply type system.

## Type system notes

Syntactically, we have a quite simply type system.

That is, our objects' types are simple.

```
lambda (x Int): x+1 # has type (Int) -> Int
fun f(v: [Int]) { v[0] == 0 } // has type [Int] -> Boolean
```

## Type system notes

Syntactically, we have a quite simply type system.
That is, our objects' types are simple.

```
lambda (x Int): x+1 # has type (Int) -> Int
fun f(v: [Int]) { v[0] == 0 } // has type [Int] -> Boolean
```

This is only half of the picture: to make our language array-oriented, we should allow arrays to be used in flexible ways.

```
Coord := { x: Float, y: Float }
fn Cx(Coord[] c) -> Float[] := c.x
```

## Typed AST

## Don’t we have types already?

What types are we missing?

## Don’t we have types already?

What types are we missing?

Expressions currently do not know their own type. This is:

- Needed to check for type-correctness anyway.
- Needed to correctly execute shape-polymorphism.

## How do we deduce types?

Recall the expressions we have:

- Literals
- Block expressions
- If-else
- Variable access
- Field access
- Function / operator calls

## Easy cases

The following are easy:

- Literals
  - Primitive type literals: already know their type
  - Record literals: already know their type
  - Lambdas: already know their parameter types, deduce return type
  - Array literals: deduce element types, check they're equal
- Blocks: deduce last expression's type
- If-else: deduce component types
- Variable access: already know their type

## Deducing field access type

Suppose we have an expression `e.a`, how do we know its type?

## Deducing field access type

Suppose we have an expression `e.a`, how do we know its type?

1. Deduce the type `R^n` of `e`.
2. Find the list of fields of `R`.
3. Find `x: T^m` in this list.
4. Conclude that the type is `T^(m+n)`.

## Deducing function and operator call types

Functions and operators are even harder to deduce, since there may be dimension mismatches.

## Deducing function and operator call types

Functions and operators are even harder to deduce, since there may be dimension mismatches. Given an expression `(f e1 e2)`:

1. Deduce the type `((T1^n1, T2^n2) -> R^k)^m` of `f`.
2. Deduce types `T1^m1` and `T2^m2` of `e1` and `e2`.
3. Decide how to broadcast.

There are different reasonable options: document your choice.

## Runtime

## Standard library

Two main concerns:

- How to use the built-in types (primitives, arrays).
- How to interact with the world.

## Standard library

Two main concerns:

- How to use the built-in types (primitives, arrays).
- How to interact with the world.

I will provide a list of functions that need to be supported, but briefly:

Numeric types:

- Arithmetic
- Comparison

Arrays:

- Size, index
- Concatenate
- Map, filter

Input/output:

- Read character
- Write character

Feel free to add more for testing.

## Next steps

## Teamwork

The first deadline is 19 October. Deliverables:

- Parser
- Pretty-printer
- Type inference
- Evaluator
- Standard library
- Test suite

Make sure to sign up your team in the spreadsheet:

- One row per team.
- Max 5 participants per team.

## Rest of the project

For now, we're just getting the basics working. Next:

Nikita's team:

1. Define 3 different concrete syntaxes.
2. Describe how to evaluate syntax choice.
3. Implement 3 parsers + pretty printers.
4. Evaluate LLM performance.

Anton's team:

1. Specify 3 LLM-powered features.
2. Describe how to evaluate each feature.
3. Implement these features.
4. Evaluate whether they serve their function.

In the last week, both teams will present their results (15-20 minutes per team + discussion).