

# Computer Language Processing

## Lab 1

---

Noé De Santo

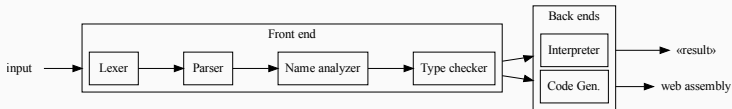
Fall 2021

# The big picture

---

# The pipeline

You will implement a full compiler/interpreter for a programming language called Amy.



# The labs

- Lab01 – Interpreter;
- Lab02 – Lexer;
- Lab03 – Parser;
- Lab04 – Type Checker;
- Lab05 – Codegen (Code Generator);
- Lab06 – Compiler extension.

# The interpreter

---

# Interpreting source code

From:

```
object Hello  
  Std.printlnString("Hello " ++ "world!")  
end Hello
```

To:

Hello World!

# Interpreting source code

From:

```
object Hello  
  Std.printlnString("Hello " ++ "world!")  
end Hello
```

To:

Hello World!

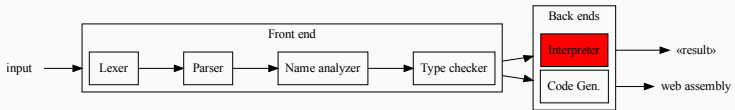
But that would be a bit difficult to do at once ...

## The interpreter *phase*

---



# Lab01

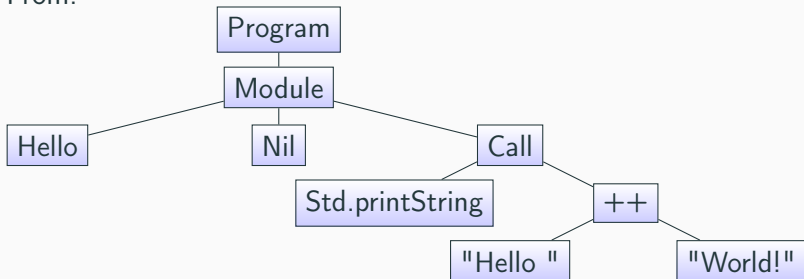


We "only" have to interpret the result of the front end.

The front end (which you will implement in labs 2,3&4) produces a data structure called an *Abstract Syntax Tree (AST)*.

# Interpreting an AST

From:



To:

Hello World!

An approximate definition of the AST is available on gitlab<sup>1</sup>.

---

<sup>1</sup>[https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/labs01\\_material/SymbolicTreeModule.scala](https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/labs01_material/SymbolicTreeModule.scala)

## Doing the lab

---

# The interpret function

You have to complete (in `src/amyc/interpreter/Interpreter.scala`)

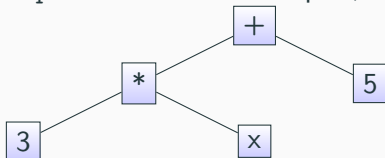
```
def interpret(expr: Expr)(implicit locals: Map[Identifier, Value]): Value
```

# The interpret function

You have to complete (in `src/amyc/interpreter/Interpreter.scala`)

```
def interpret(expr: Expr)(implicit locals: Map[Identifier, Value]): Value
```

- `expr` is the AST to interpret, e.g.

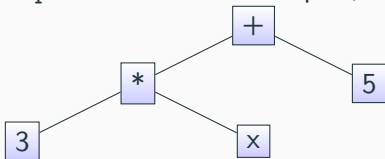


# The interpret function

You have to complete (in `src/amyc/interpreter/Interpreter.scala`)

```
def interpret(expr: Expr)(implicit locals: Map[Identifier, Value]): Value
```

- `expr` is the AST to interpret, e.g.



- `locals` maps variable identifiers into their values, e.g.

```
{  
  x -> 8,  
  bestFruit -> "Tomato"  
}
```

- Locate the ???s:

```
case Times(lhs, rhs) => ???
```

- Locate the ???s:

```
case Times(lhs, rhs) => ???
```

- Refer to the amy specification<sup>2</sup> (section 3.5) for the expected behavior of the expression:  
« +, -, \*, / and % have type (Int, Int)  $\Rightarrow$  Int, and are the usual integer operators. »

---

<sup>2</sup>[https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/amy\\_specification.pdf](https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/amy_specification.pdf)



# Workflow

- Locate the ???s:

```
case Times(lhs, rhs) => ???
```

- Refer to the amy specification<sup>2</sup> (section 3.5) for the expected behavior of the expression:

*« +, -, \*, / and % have type (Int, Int)  $\Rightarrow$  Int, and are the usual integer operators. »*

- Implement the required semantic:

```
case Times(lhs, rhs) => IntValue(  
  interpret(lhs).asInt *  
  interpret(rhs).asInt  
)
```

---

<sup>2</sup>[https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/amy\\_specification.pdf](https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/amy_specification.pdf)

# The front end is nice

The front end will reject non-sensical programs

# The front end is nice

The front end will reject non-sensical programs<sup>3</sup>

```
object Bogus  
  "Amy <3" || 5  
end Bogus
```

---

<sup>3</sup>Useless fact: "Amy <3" || 5 is valid in javascript; it evaluates to "Amy <3"

# The front end is nice

The front end will reject non-sensical programs<sup>3</sup>

```
object Bogus
```

```
    "Amy <3" || 5
```

```
end Bogus
```

```
sbt:amyc> run examples/Bogus.scala
```

```
[info] running amyc.Main examples/Bogus.scala
```

```
[ Error ] examples/Bogus.scala:2:17: Type error: expected Boolean, found String
```

```
[ Error ]           "Amy <3" || 5
```

```
[ Error ]           ^
```

```
[ Error ] examples/Bogus.scala:2:29: Type error: expected Boolean, found Int
```

```
[ Error ]           "Amy <3" || 5
```

```
[ Error ]           ^
```

```
[ Fatal ] There were errors.
```

So you can assume that the AST always represents a valid program.

---

<sup>3</sup>Useless fact: "Amy <3" || 5 is valid in javascript; it evaluates to "Amy <3"

## We provide some tests...

---

```
object EmptyObject
```

```
end EmptyObject
```

---

EmptyObject.scala

---

```
object MinimalError
```

```
    error("")
```

```
end MinimalError
```

---

MinimalError.scala

## We provide some tests...

---

```
object EmptyObject
```

```
end EmptyObject
```

---

EmptyObject.scala

---

```
object MinimalError
```

```
    error("")
```

```
end MinimalError
```

---

MinimalError.scala

...but you should write your own.

*«If I were to take something from this course, it would be learning to write good tests»*

Rodrigo

# Tips and tricks

- Read The Fine Manual: the specification<sup>4</sup> contains every information you need:
  - Section 1 explains most features of Amy;
  - Section 3 contains crucial details for this assignment;
  - Reading section 2 might also help you understand some intricacies of Amy;
- Even though the file is not included in the skeleton, `SymbolicTreeModule.scala`<sup>5</sup> is useful to know what the fields of the different nodes of the AST are;
- The handout and the comments contain some additional hints on how to implement some of the most difficult parts;
- You can run examples/tests even with an incomplete interpreter.

---

<sup>4</sup>[https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/amy\\_specification.pdf](https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/amy_specification.pdf)

<sup>5</sup>[https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/labs01\\_material/SymbolicTreeModule.scala](https://gitlab.epfl.ch/lara/cs320/-/blob/main/labs/labs01_material/SymbolicTreeModule.scala)



**Good Luck !**