

# Computer Language Processing

## Lab 3

---

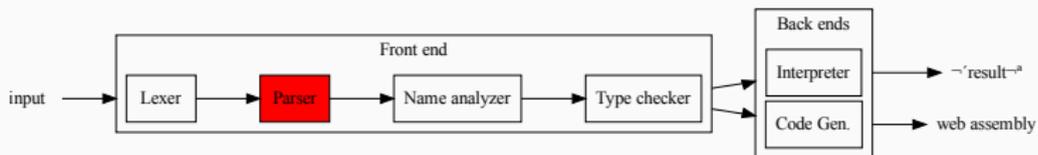
Julie Giunta

Fall 2022

# The labs

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

# Lab03

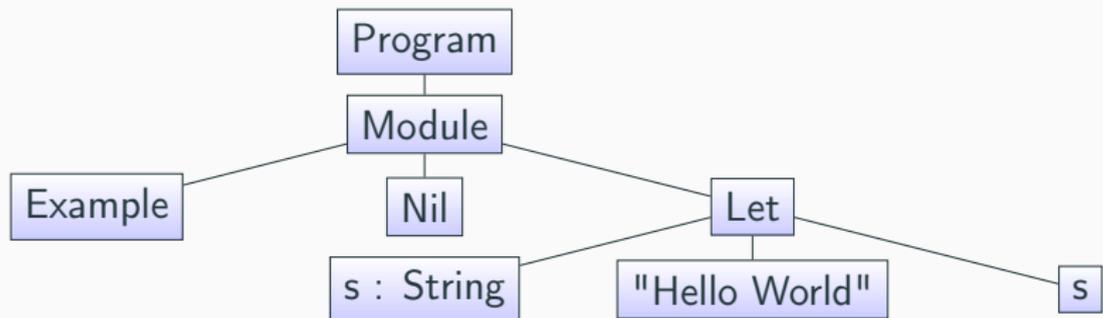


You will have to transform a sequence of tokens to an AST (Abstract Syntax Tree).

# Program

```
object Example
  val s: String = "Hello world";
  s
end Example
```

```
KeywordToken(object) (1:1)
IdentifierToken(Example) (2:8)
KeywordToken(val) (2:1)
IdentifierToken(s) (2:5)
DelimiterToken(:) (2:6)
PrimTypeToken(String) (2:8)
OperatorToken(=) (2:15)
StringLitToken(Hello world) (2:17)
DelimiterToken(;) (2:30)
IdentifierToken(s) (3:1)
KeywordToken(end) (4:1)
IdentifierToken(Example) (4:5)
EOFToken() (4:12)
```



# Grammar

```
Program ::= Module*
Module ::= object Id Definition* Expr? end Id
Definition ::= AbstractClassDef | CaseClassDef | FunDef
AbstractClassDef ::= abstract class Id
CaseClassDef ::= case class Id ( Params ) extends Id
FunDef ::= fn Id ( Params ) : Type = { Expr }
Params ::= ε | ParamDef [ , ParamDef ]*
ParamDef ::= Id : Type
Type ::= Int ( 32 ) | String | Boolean | Unit | [ Id . ]? Id
Expr ::=
| Literal
| Expr BinOp Expr
| UnaryOp Expr
| [ Id . ]? Id ( Args )
| Expr ; Expr
| val ParamDef = Expr ; Expr
| if ( Expr ) { Expr } else { Expr }
| Expr match { MatchCase+ }
| error ( Expr )
| ( Expr )
Literal ::= true | false | ( )
BinOp ::= + | - | * | / | % | < | <=
| && | || | == | ++
UnaryOp ::= - | !
MatchCase ::= case Pattern => Expr
Pattern ::= [ Id . ]? Id ( Patterns ) | Id | Literal | _
Patterns ::= ε | Pattern [ , Pattern ]*
Args ::= ε | Expr [ , Expr ]*
```

# Implementation

---

# Implementation

- You have to replace the ??? in the file `src/amyc/parsing/Parser.scala`
- You will use Scallion (See [this introduction to Scallion parser combinators.](#))
- You will have to encode Amy's grammar so that it is LL(1) (See [this lecture](#)).

# Tree Module

- You will transform sequences of tokens to trees of the `NominalTreeModule`. (See `src/amyc/ast/TreeModule.scala`)
- Names have not been resolved yet. They will be resolved during the name analysis phase where the tree will be transformed into a `SymbolicTreeModule`.
- Currently, `Name` is a `String` and `QualifiedName` a pair of `Strings`.
- After the name analysis, they will be transformed into unique identifier. Do not worry about that yet, there will be plenty of time during next lab ;)

- `elem(kind)` : takes the kind of tokens accepted and produces a `Parser[Token]`.
- `accept(kind)` : applies directly a transformation to the tokens that are accepted.
- `epsilon(value)` :  $\epsilon$  in grammars

## Scallion parser combinators

- $p1 \mid p2$  : disjunction
- $p1 \sim p2$  : sequence
- `map` : applies a transformation to the values produced by a parser.
- `recursive` : needed when you want to recursively invoke your parser.
- `opt` : mark a parser optional.
- `many`, `many1` : accepts a number of repetitions of its argument parser
- `repsep`, `rep1sep` : accepts a number of repetitions of its argument parser, separated by an other parser.
- `Operators trait` : for infix binary operators, with different associativities and priority levels

See [Noé's introduction](#) to learn more about Scallion.

## Example

```
Module ::= object Id Definition* Expr? end Id
```

```
lazy val module =  
  (kw("object") ~ identifier ~ many(definition) ~  
    opt(expr) ~ kw("end") ~ identifier).map {  
case obj ~ id ~ defs ~ body ~ _ ~ id1 =>  
  if id == id1 then  
    ModuleDef(id, defs.toList, body).setPos(obj)  
  else  
    throw new AmycFatalError("Error")  
}
```

## Example

```
def kw(string: String): Syntax[Token] =  
    elem(KeywordKind(string))  
  
val identifier: Syntax[String] = accept(IdentifierKind) {  
    case IdentifierToken(name) => name  
}  
  
lazy val definition: Syntax[ClassOrFunDef] = ???  
lazy val expr: Syntax[Expr] = recursive { ??? }
```

- Read carefully the assignment.
- Have a look at the new/updated files for this assignment.
- Do not forget to call *setPos* to set the position of the nodes of the AST.
- Write as many tests as possible.

**Good Luck !**