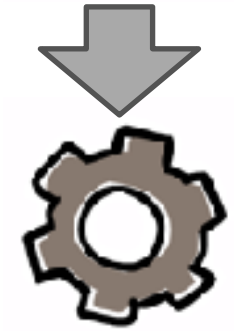
THE FOOL LANGUAGE

Cosimo Laneve

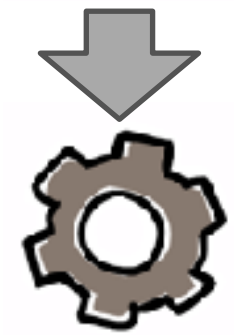
corso di **Compilatori & Interpreti**

this lecture

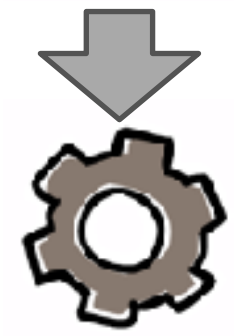
lexical analysis



syntactic analysis



semantic analysis



bytecode generation

the FOOL interpreter

FOOL

- = **F**unctional **O**bject-**O**riented **L**anguage
- is an imperative language with two data-types (**`int`** and **`bool`**)
  - standard assignment statement `x = exp ;`
- it admits variable declarations
  - standard declaration `let int x = 4 ; in x+1`
- it admits function definitions
  - standard definition `let int foo(int x) x+1; in foo(34) ;`
- it **does not admits recursion**

FOOL — simple examples

```
let int x = 3; in print(x+2);
```

- prints 5

```
let int x = 2 ; in print(x==3);
```

- prints `false`

```
let bool foo(int x)
   if (x==0) then { true }
   else { false)}
in print(foo(3));
```

- prints `false`

FOOL — extras

- `let` are top-level or inside bodies of functions
- examples

```
let int x = 1; bool b = true;
in print (if (b) then { x+1 } else { x+2 });
```

- other examples

```
let int foo(int n)
    let int gee(int n) n+1 ; in
    if (n==0) then { 1 } else { n*gee(n - 1) } ;
in print(foo(3)) ;
```

```
let int gee(int n) n+1 ;
    int foo(int n)
        if (n==0) then { 1 } else { n*gee(n - 1) } ;
in print(foo(3)) ;
```

  - are equal

ANTLR — the Simple example

```
grammar Simple;
block     : '{' statement* '}';
statement : assignment ';'
          | deletion ';'
          | print ';'
          | block;
assignment    : ID '=' exp;
deletion  : 'delete' ID;
print     : 'print' exp;
exp           : '(' exp ')'                          #baseExp
          | '-' exp                                  #negExp
          | left=exp op=('*' | '/') right=exp        #binExp
          | left=exp op=('+' | '-') right=exp        #binExp
          | ID                                       #varExp
          | NUMBER                                   #valExp;
//IDs
fragment CHAR     : 'a'..'z' |'A'..'Z' ;
ID              : CHAR (CHAR | DIGIT)* ;
//Numbers
fragment DIGIT    : '0'..'9';
NUMBER          : DIGIT+;
//SCAPED SEQUENCES
WS              : (' '|'\t'|'\n'|'\r')-> skip;
LINECOMENTS   : '//' (~('\n'|'\r'))* -> skip;
BLOCKCOMENTS    : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'|BLOCKCOMENTS)* '*/' -> skip;
```

the node in the syntax tree is not "exp" but what is specified by #

no node in the syntax tree is generated! CHAR and DIGIT are collected in the tokens ID and NUMBER (they are not referenced from parser's rules)

no node in the syntax tree is generated! The characters are skipped

next lecture