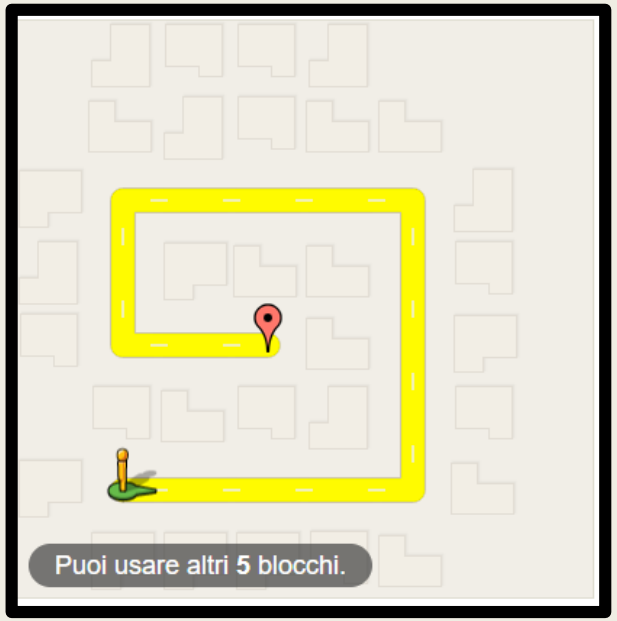
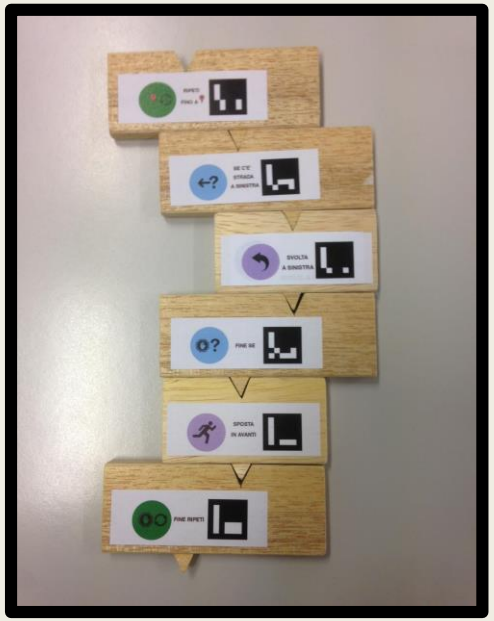
RECURSIVE DESCENT PARSER IN CODOWOOD

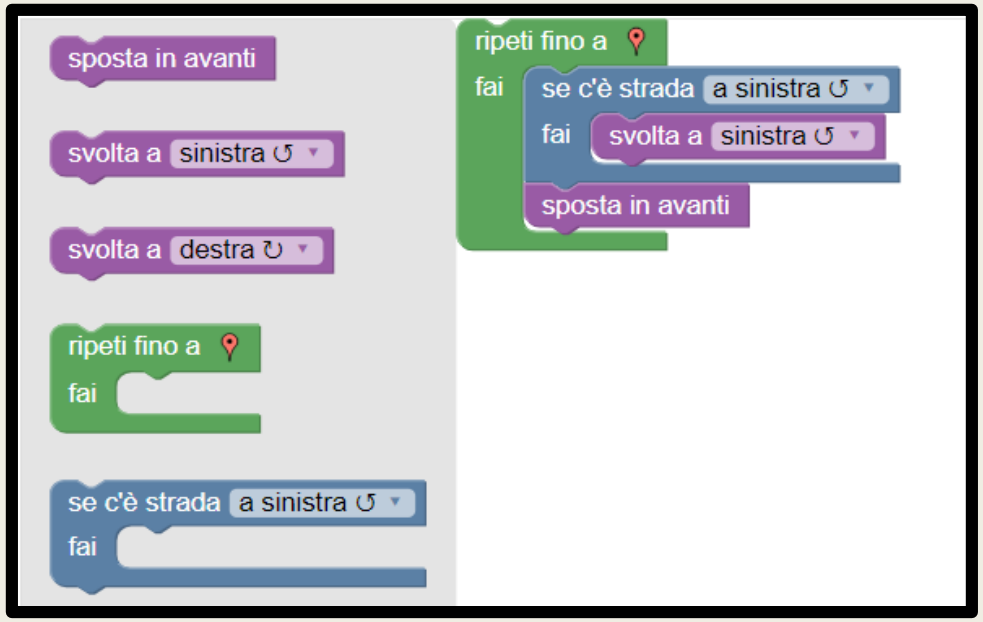
Gianfranco Lombardo

CodOWood

- Tangible programming framework, developed in SoWide with the aim of introduce children to the «Computational thinking»
- Usage: Using different kind of wood blocks, it is possible to define the actions that a charachter has to follow in order to get out of the maze.

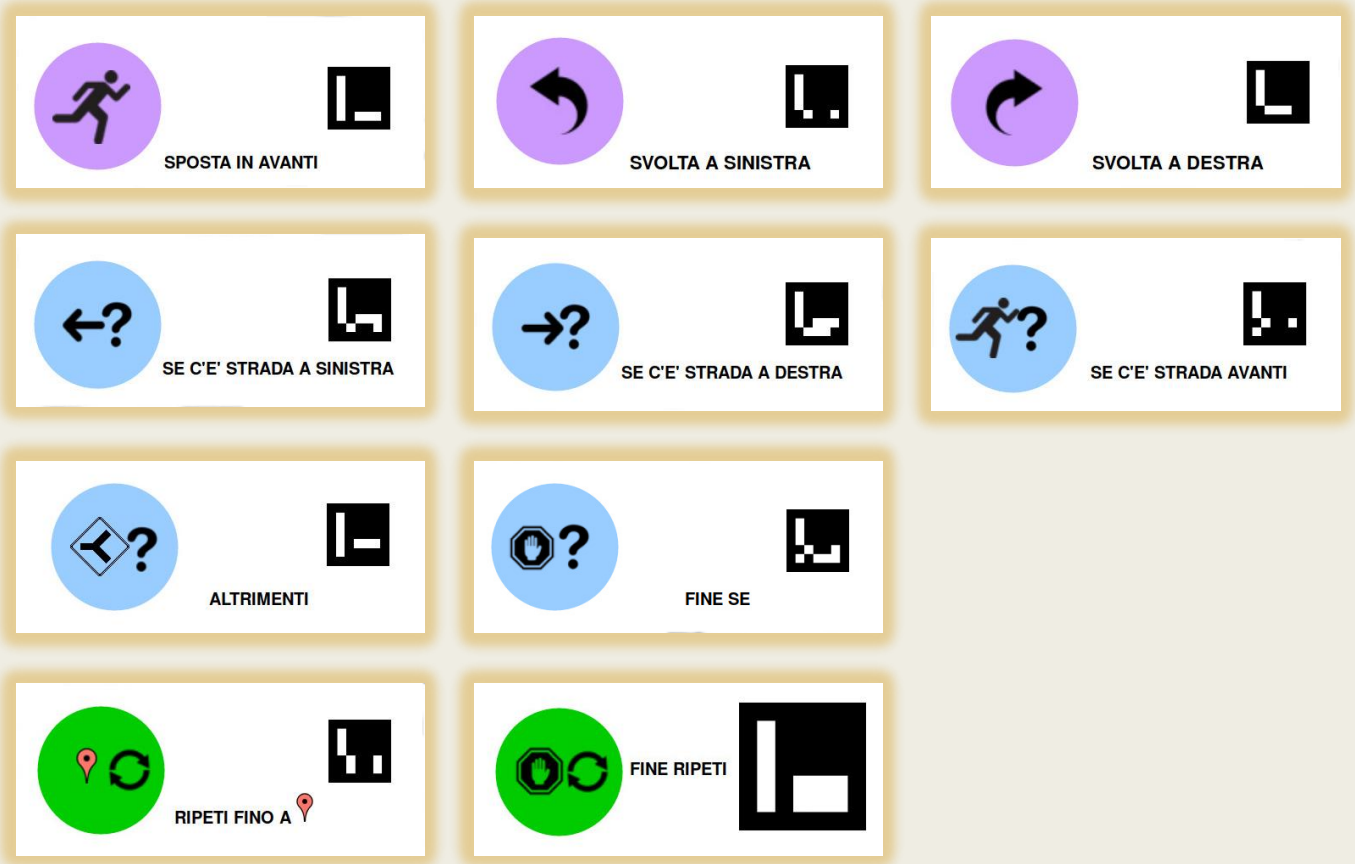






Language

- Each wood block get translated in an equivalent virtual block defined using Blockly (Google)
- Each block represents an instruction



Blocchi corrispondenti definiti utilizzando Blockly

Language

- Each wood block get recognised using image processing algorithms and after that it being translated into the equivalent instruction

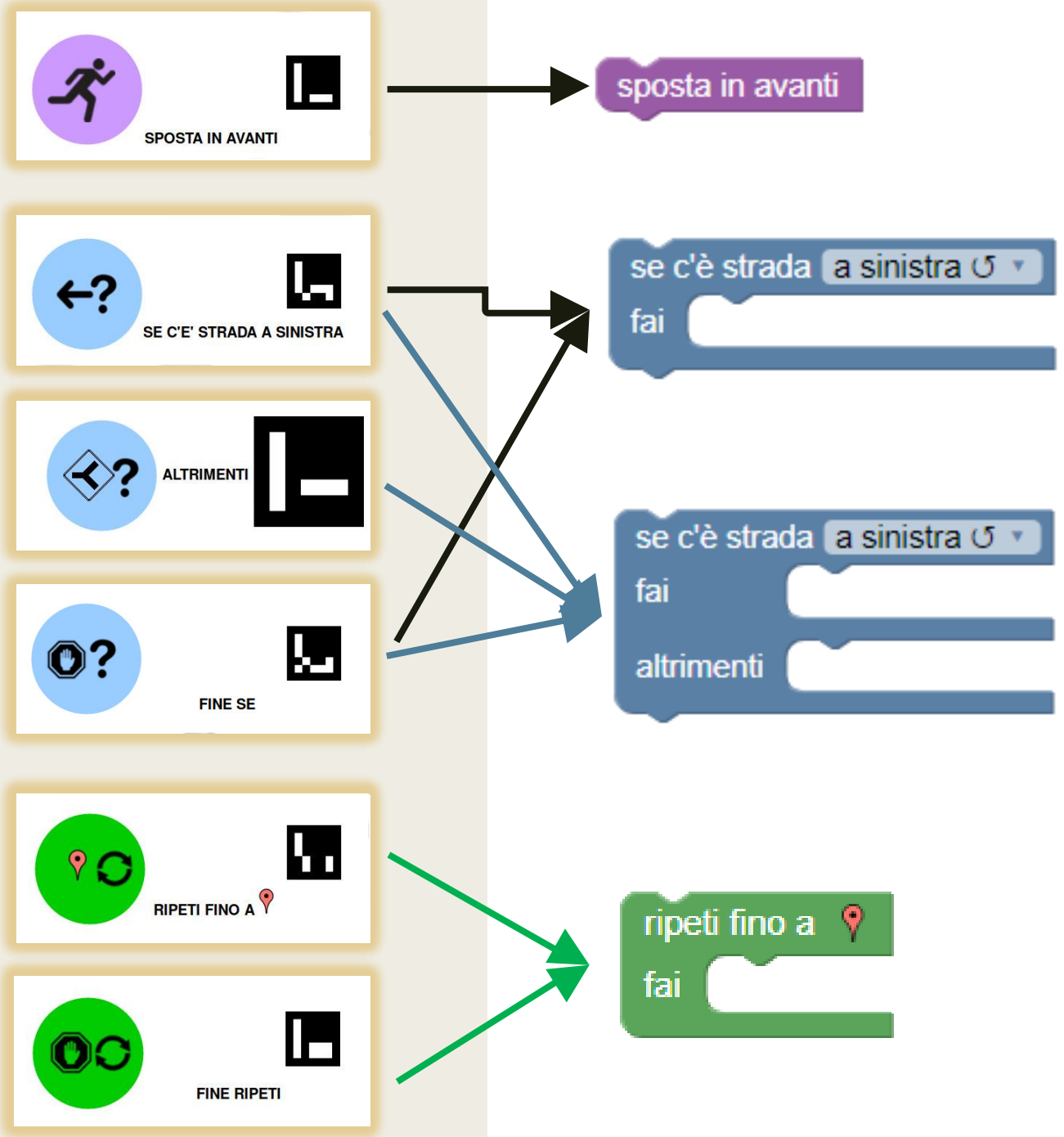


Photo of the «wood solution»

Image processing

[1,5,2,4,7]

The identified sequence of instructions

Description of the instructions in the XML format

Blockly

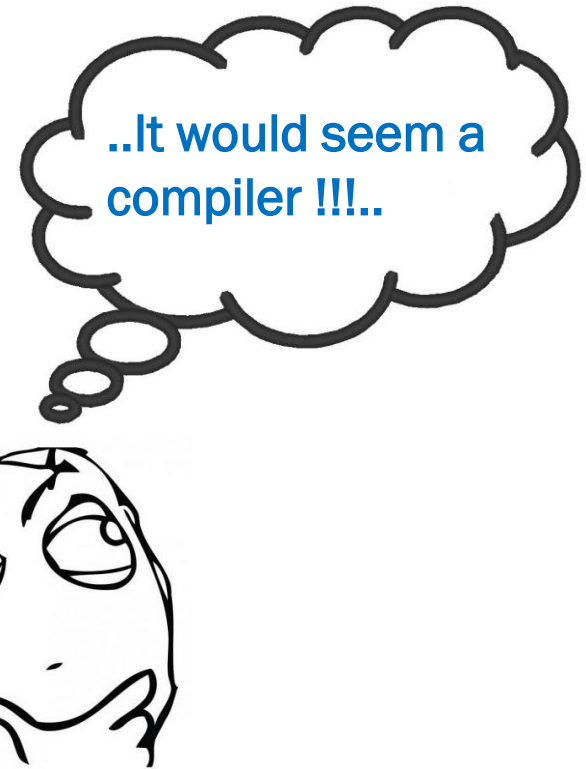
Sequence requested by CoodOWood

[1,5,2,4,7]

The identified sequence of instructions

- Sintattic analysis
- Semantic analysis
- Conversion of the input in the requested XML Blockly format
- Output synthesis

Description of the instructions in the XML format



Source code

[1,5,2,4,7]

The identified sequence of instructions

- Sintattic analysis
- Semantic analysis
- Conversion of the input in the requested XML Blockly format
- Output synthesis

Description of the instructions in the XML format

Object file

Grammar LL(1)

- The «source code» is a list of tokens that is entirely available to the next step of the processing

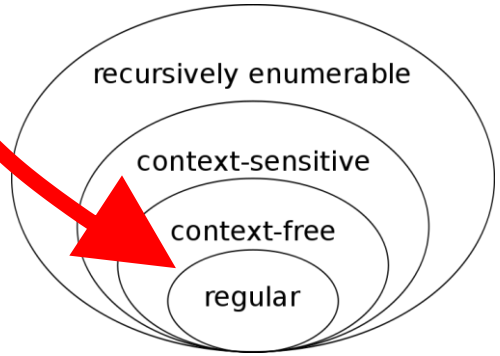
- «Peeking» of the next token is available, so no Backtracking is needed
- For semplicity k=1

Sequence of the available actions (Tokens for us):

- **GO_FWD** = 1
- **TURN_LEFT** = 2
- **TURN_RIGHT** = 3
- **ELSE** = 4
- **END_LOOP** = 5
- **IF_LEFT** = 6
- **IF_RIGHT** = 7
- **IF_FWD** = 8
- **END_IF** = 9
- **LOOP** = 10

Context-free grammar Review

- A -> δ , A Є VN , δ Є (VT U VN)
- No unreachable symbols
- No unproductive symbols
- No cycles



Chomsky Hierarchy

- Type-0 : α -> β with no restrictions
- Type-1: αAβ -> α δ β
- Type-3: A -> α and A -> α B

Let's build our grammar !!!

Grammar Review

- G = (VT, VTN, P, S)
  - *VT : Terminal symbols*
  - *VN : Nonterminal symbols*
  - *P: Production rules*
  - *S : Start symbol (or sentence symbol)*

Our VT:

("Go_forward","Turn_left","Turn_right",
"Loop","End_Loop","If_forward",
"If_left","If_right","Else","End_If")

S:

sequence ::= { instruction }

Note: We are going to use the Extended Backus Naur Form (EBNF) Notation.
In EBNF two { .. } specify the «repeat symbol» concept

CodOWood Grammar

VT :

(“Go_forward”,“Turn_left”,“Turn_right”, “Loop”,“End_Loop”,“If_forward”, “If_left”,“If_right”,“Else”,“End_If”)

Let’s try to define the Nonterminal set and the production rules with the EBNF Notation

- **sequence ::= { instruction }**

.

.

.

.

.

Recursive Descent Parser

- Top-down parser suitable for LL(k) grammar
- Built from a set of mutually recursive procedures
- No Backtracking required in the «Predictive parsing» case
- Each procedure implements one of the productions of the grammar

In depth

- Recursive descent with backtracking is possible and is not limited to LL(k) grammars
- It is not guaranteed to terminate unless the grammar is LL(k)
- Anyway it can require exponential time

Example in Python

Expression solver

- expr = term {('+' | '-') term}
- term = factor {('*' | '/') factor}
- factor = '-' factor | '(' expr ')' | var | num
- var = 'w' | 'x' | 'y' | 'z'

Expression example : (x + w) * (x + y) * (y - z)

```
def expr(..):
        .....
def term(..):
        .....
def factor(..):
        .....
def var(..):
        .....
```

Exercise: Complete grammar in CoodOWood

Production rules:

Sequence ::= { Instruction }
Instruction ::= «go forward» | «turn left» | «turn right» | Loop | If
Loop ::= «loop» Sequence «end loop»
If ::= («if forward»| «if left» | «if right») Sequence Else_end
Else_end ::= «else» Sequence «end if» | «end if»

We need to implement five functions:

- ParseSequence : ☑
- ParseInstruction : ☑
- ParseLoop : ☑
- ParseIf : TO DO
- ParseElseEnd : TO DO

Javascript method: splice ()

- In order to consume tokens in the istruction array

The splice() method adds/removes items to/from an array, and returns the removed item(s).
Note: This method changes the original array.

Syntax
array.splice(*index*, *howmany*, *item1*, ..., *itemX*)

Parameter	Description
index	Required. An integer that specifies at what position to add/remove items, Use negative values to specify the position from the end of the array
howmany	Optional. The number of items to be removed. If set to 0, no items will be removed
item1, ..., *itemX*	Optional. The new item(s) to be added to the array

```
<block type="maze_forever">
  <statement name="DO">
    <block type="maze_if">
      <field name="DIR">isPathLeft</field>
      <statement name="DO">
        <block type="maze_turn">
          <field name="DIR">turnLeft</field>
        </block>
      </statement>
      <next>
        <block type="maze_ifElse">
          <field name="DIR">isPathForward</field>
          <statement name="DO">
            <block type="maze_moveForward">
            </block>
          </statement>
          <statement name="ELSE">
            <block type="maze_turn">
              <field name="DIR">turnRight</field>
            </block>
          </statement>
        </block>
      </next>
    </block>
  </statement>
</block>
```