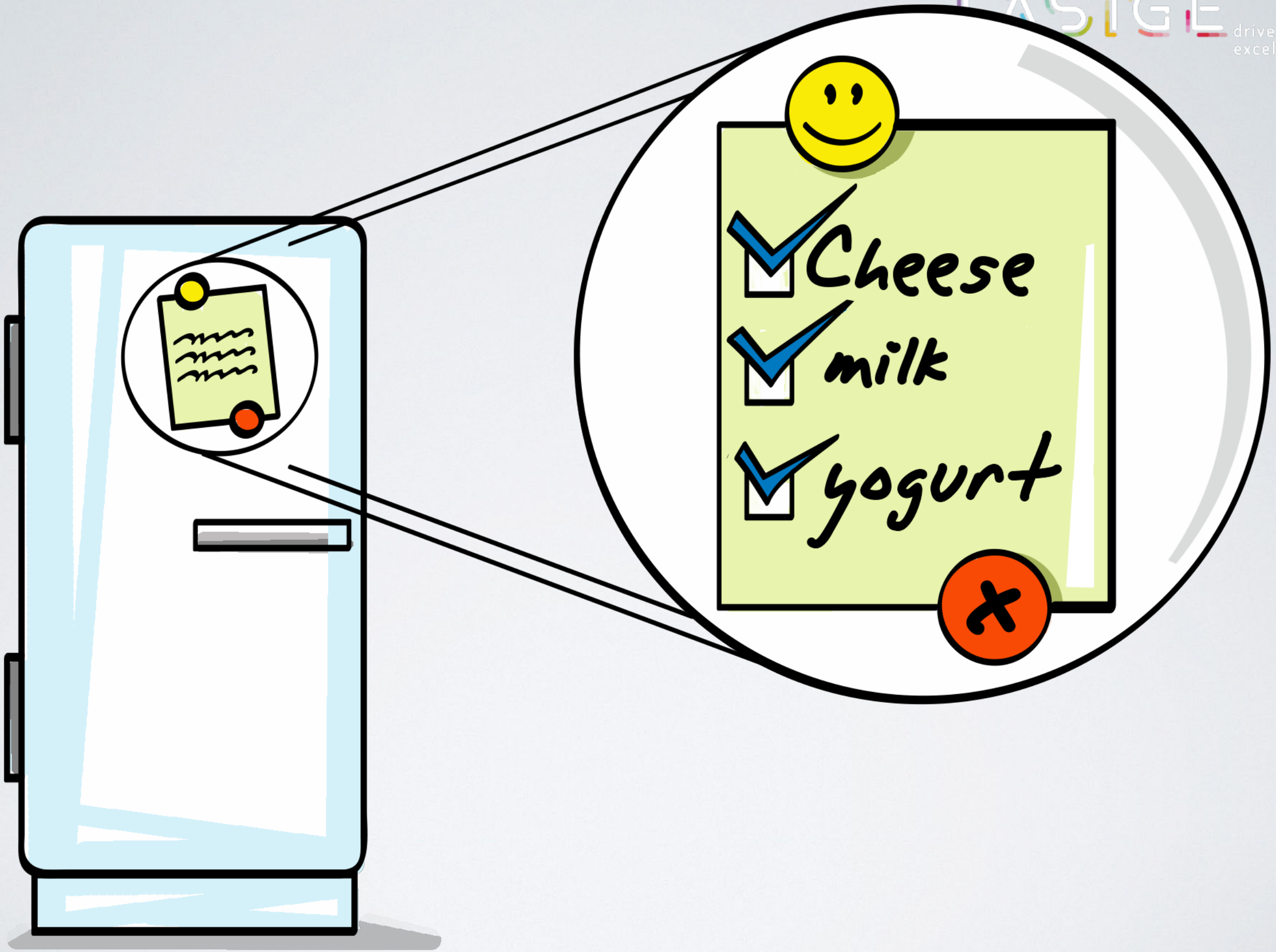
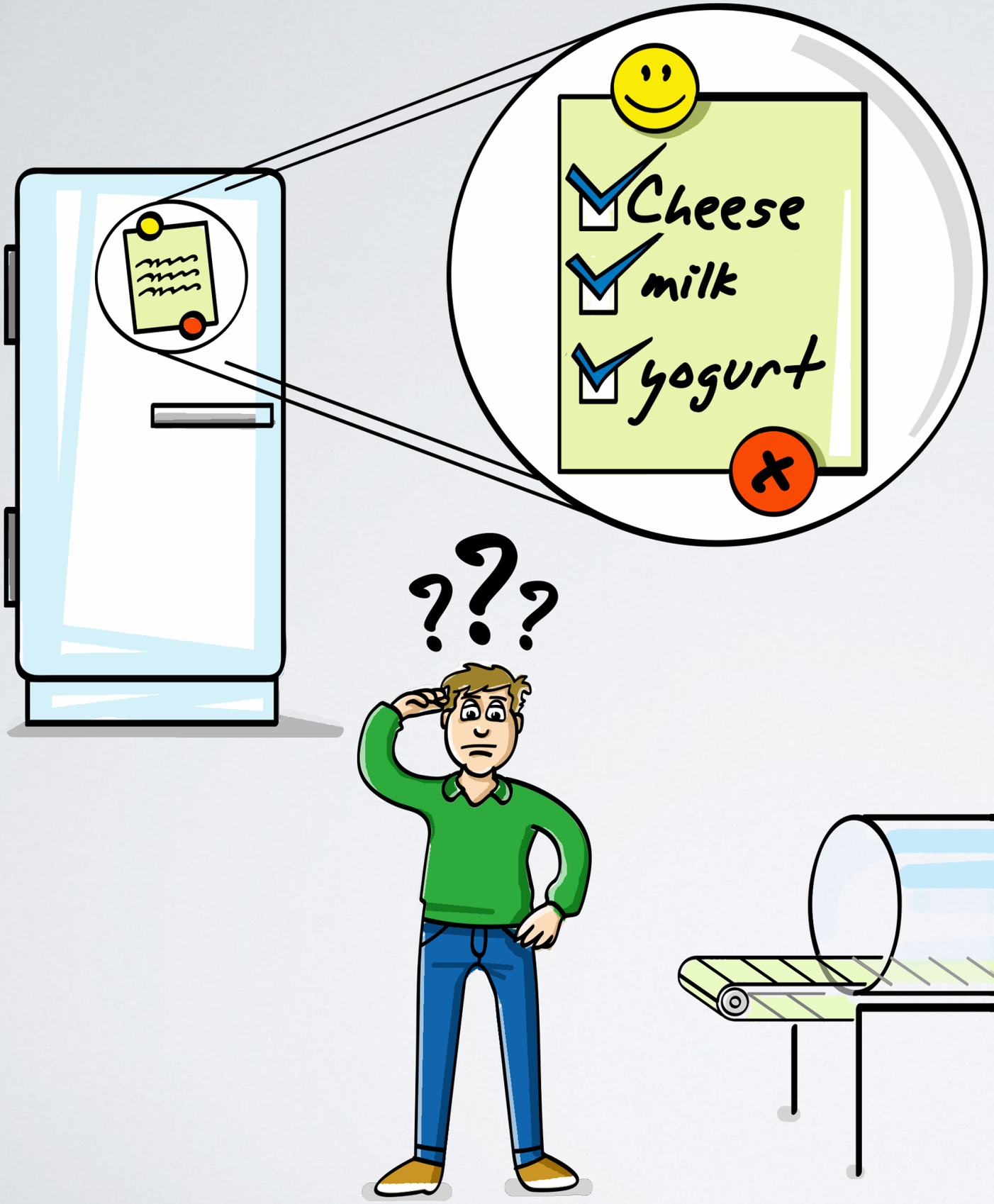
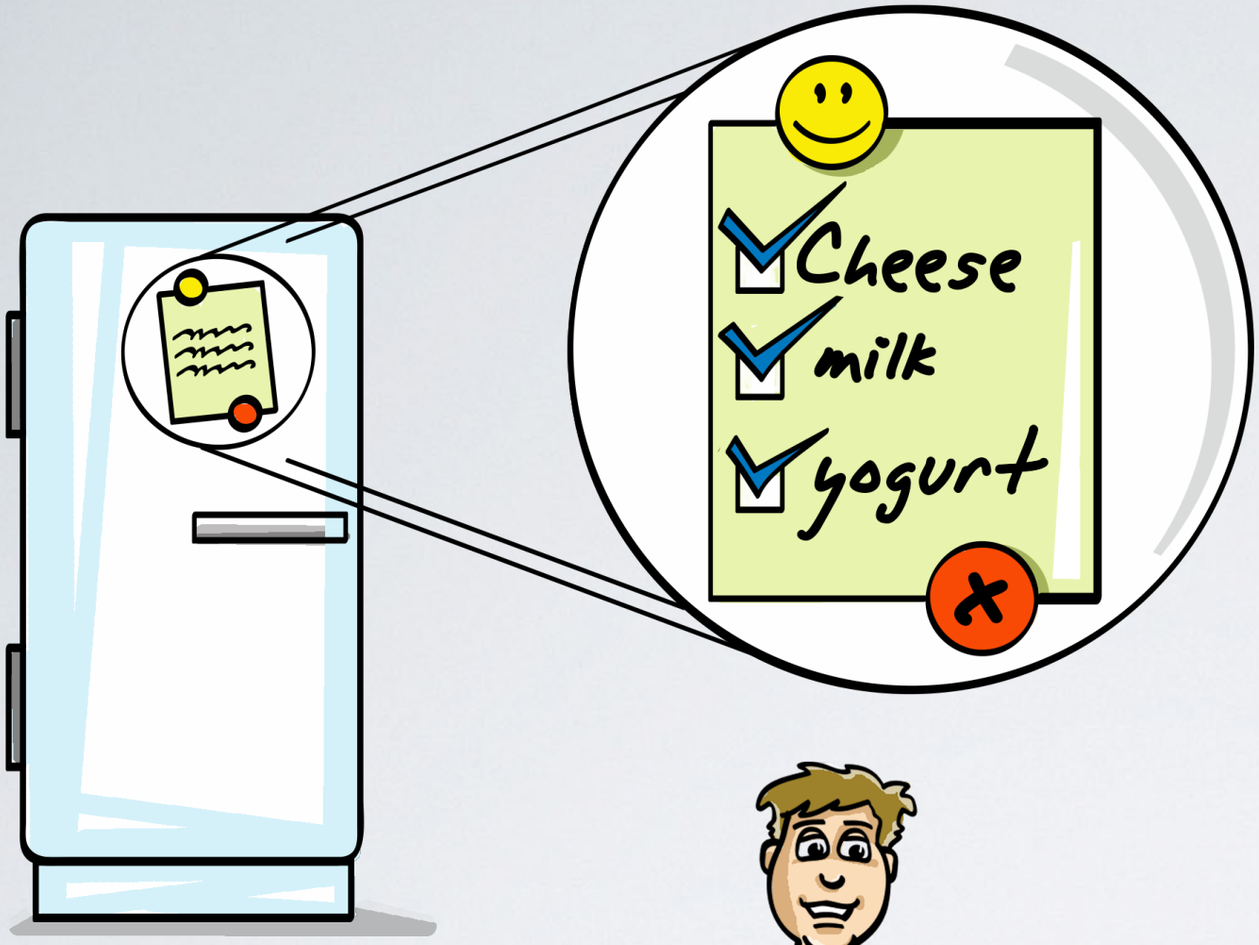
FREEST _ CONTEXT-FREE SESSION TYPES IN CONCURRENT FUNCTIONAL LANGUAGE

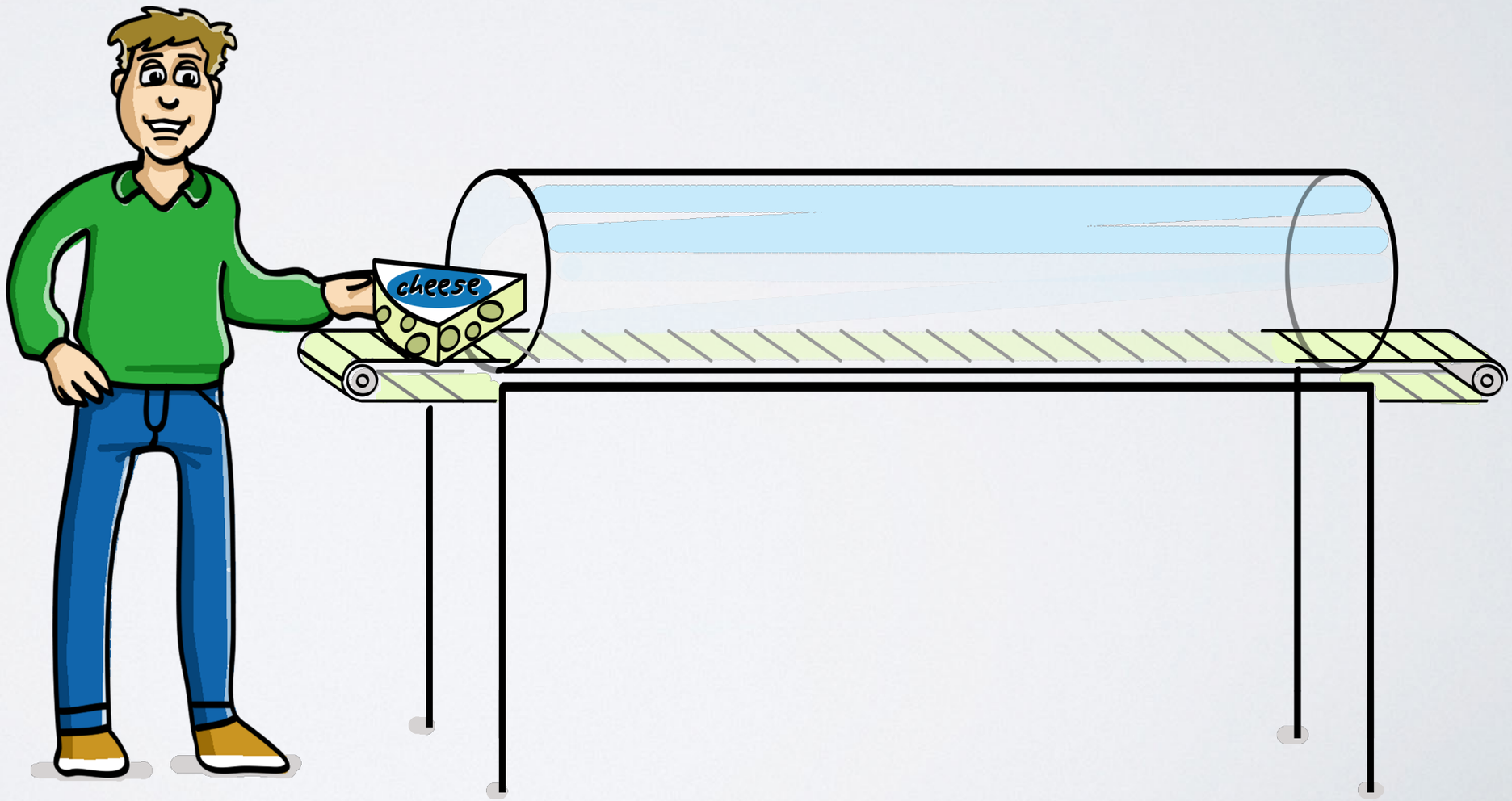
Vasco T. Vasconcelos, Bernado Almeida, Andreia Mordido
University of Lisbon

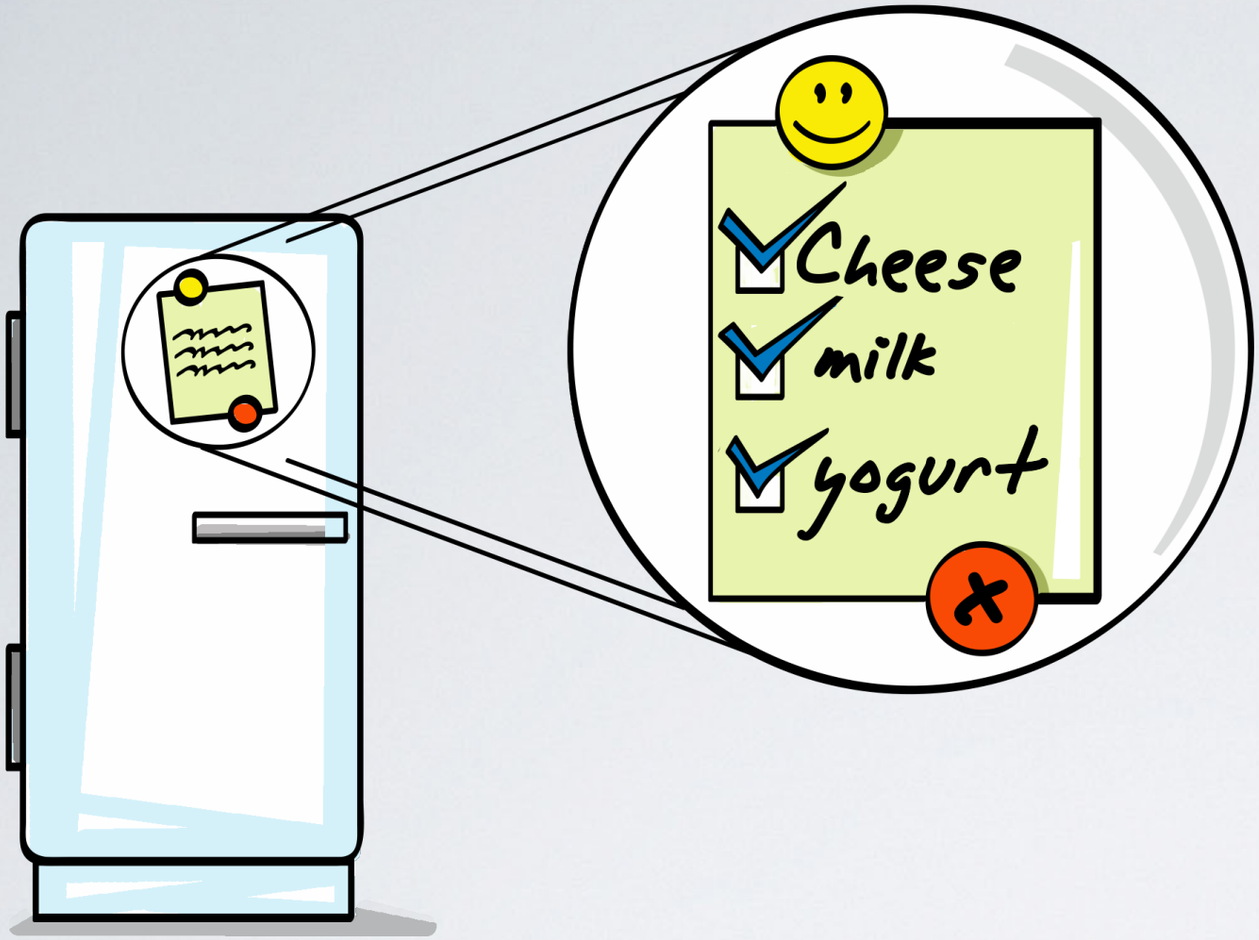
Programming Languages for Distributed Systems
Shonan, May 2019



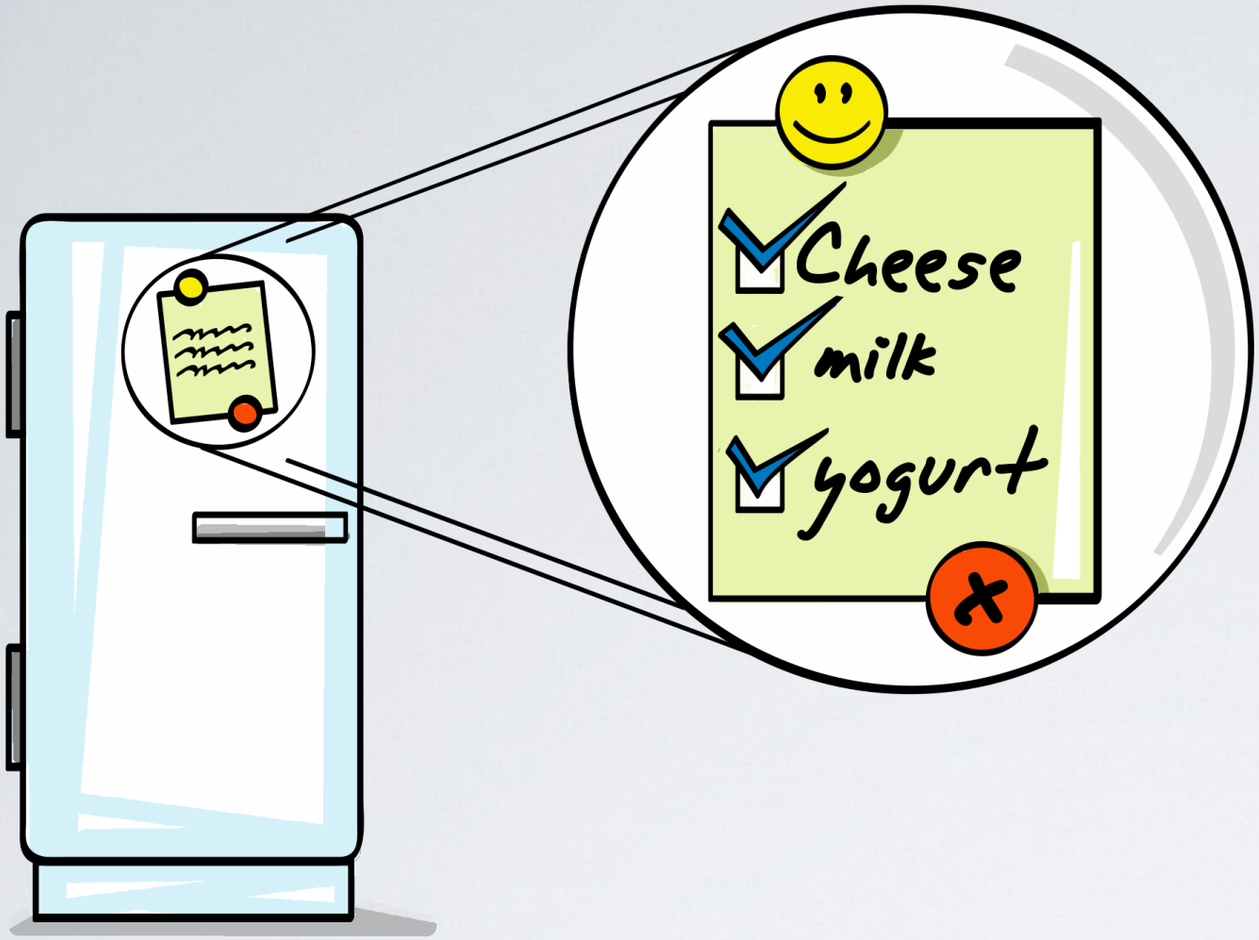

Cheese
milk
yogurt
x
???



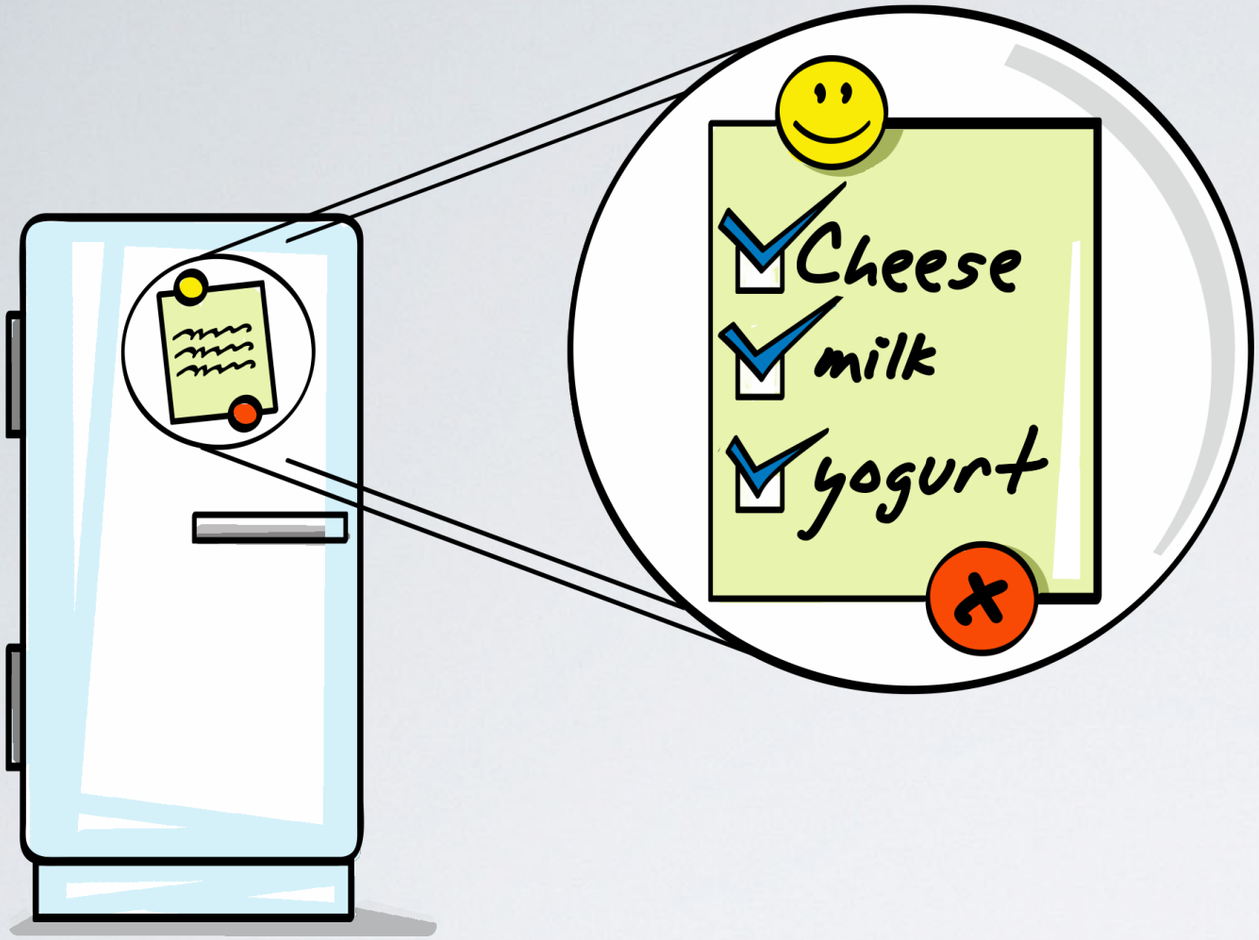



Cheese
milk
yogurt

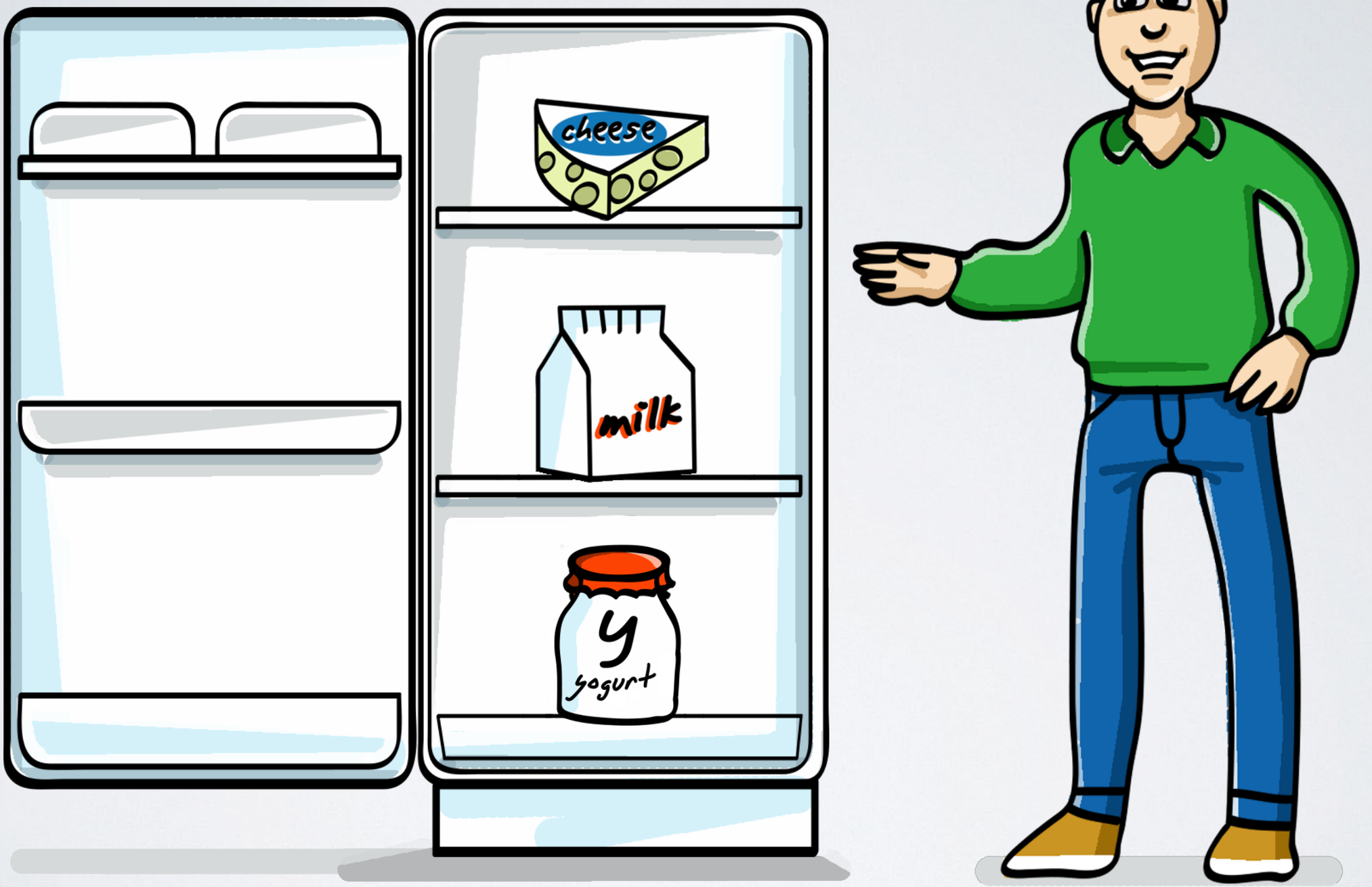

milk
cheese


Cheese
milk
yogurt


yogurt
milk
cheese


Cheese
milk
yogurt


y
yogurt
milk
cheese


cheese
milk
y
yogurt

LASIGE driven by excellence

REGULAR SESSION-TYPES

The type of the server

Choice

```
type ListServer = &{
```

Termination

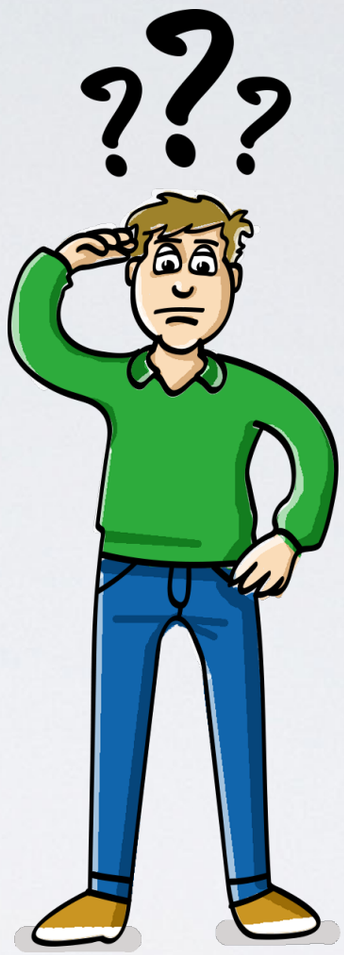
```
  Nil : end,
```

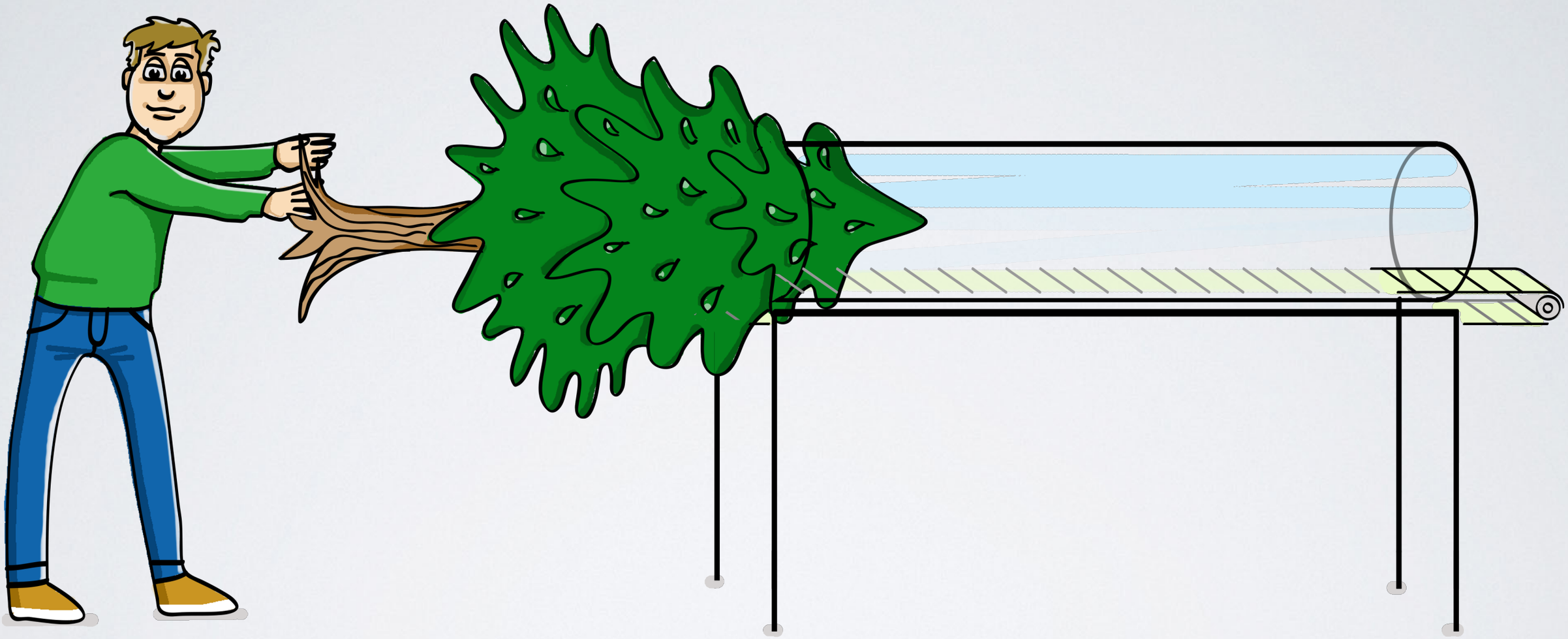
Choice label

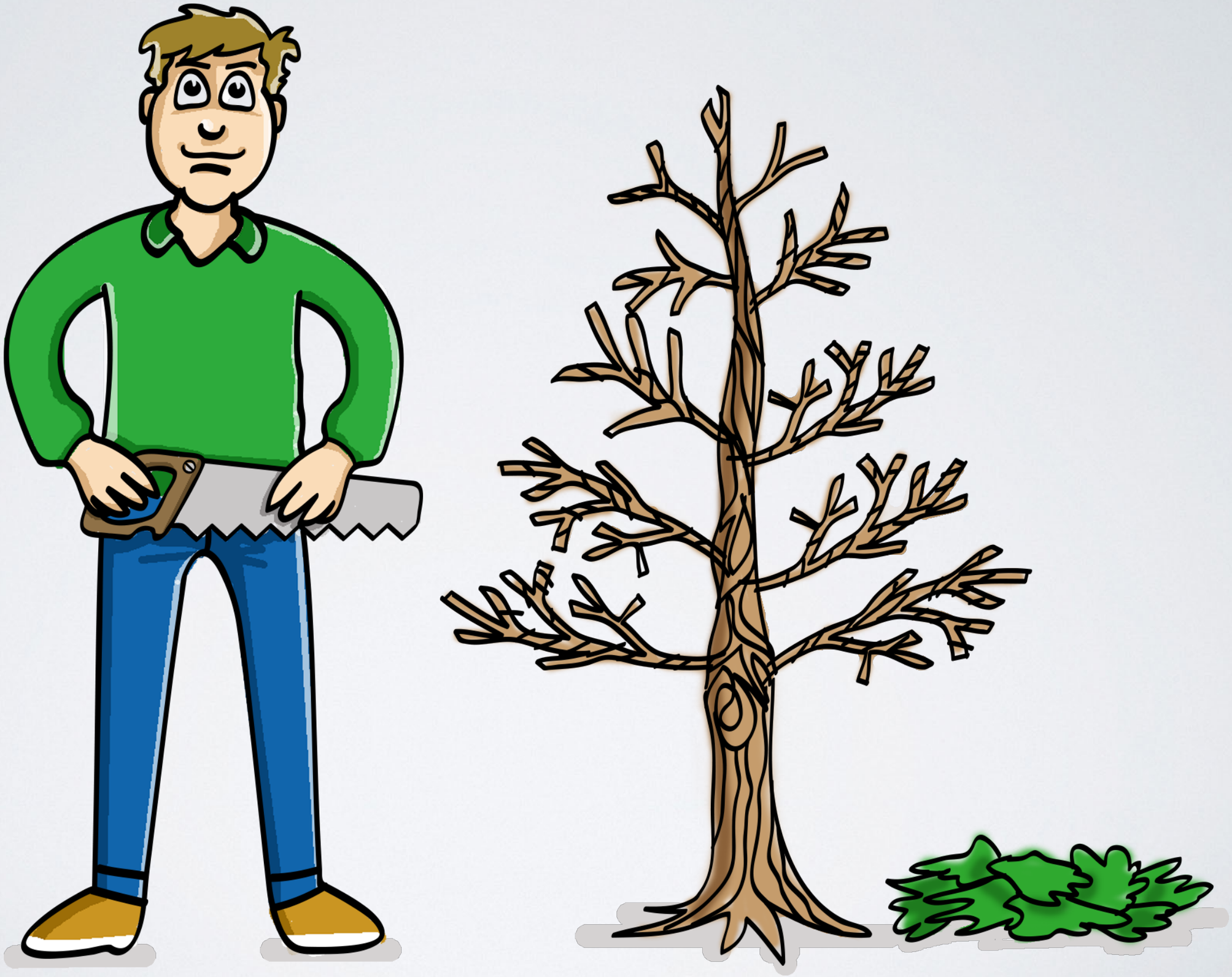
```
  Cons: ?Int . ListServer
}
```

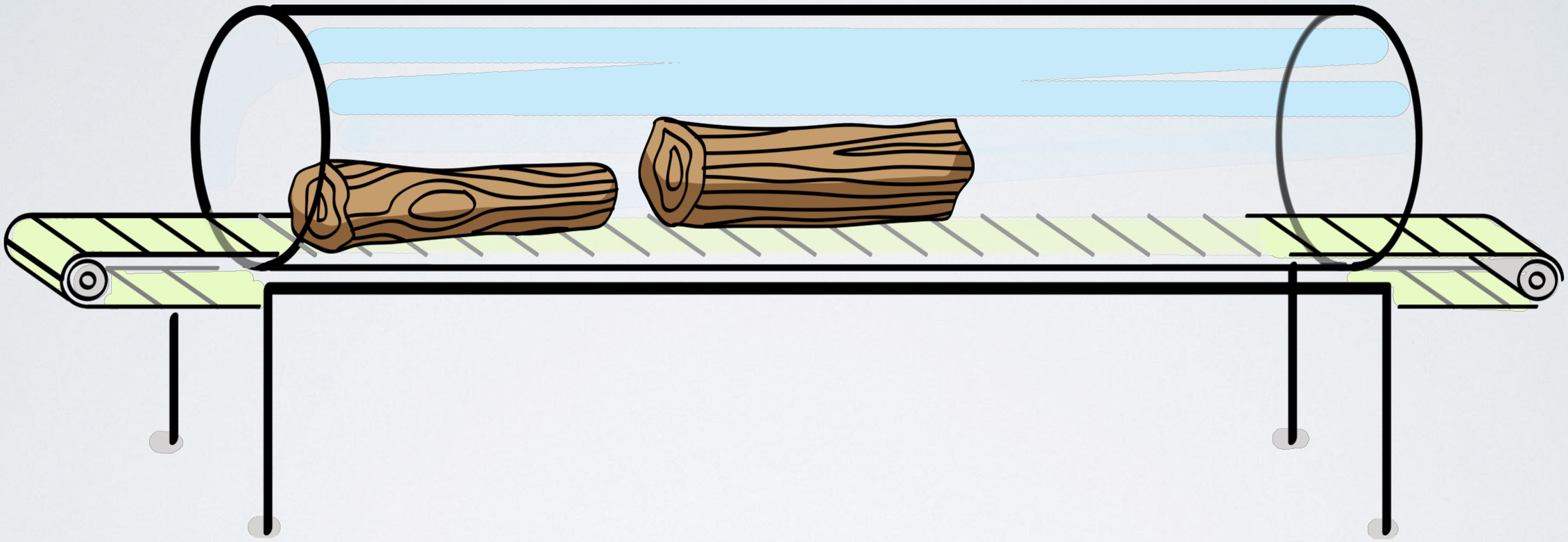
Receive

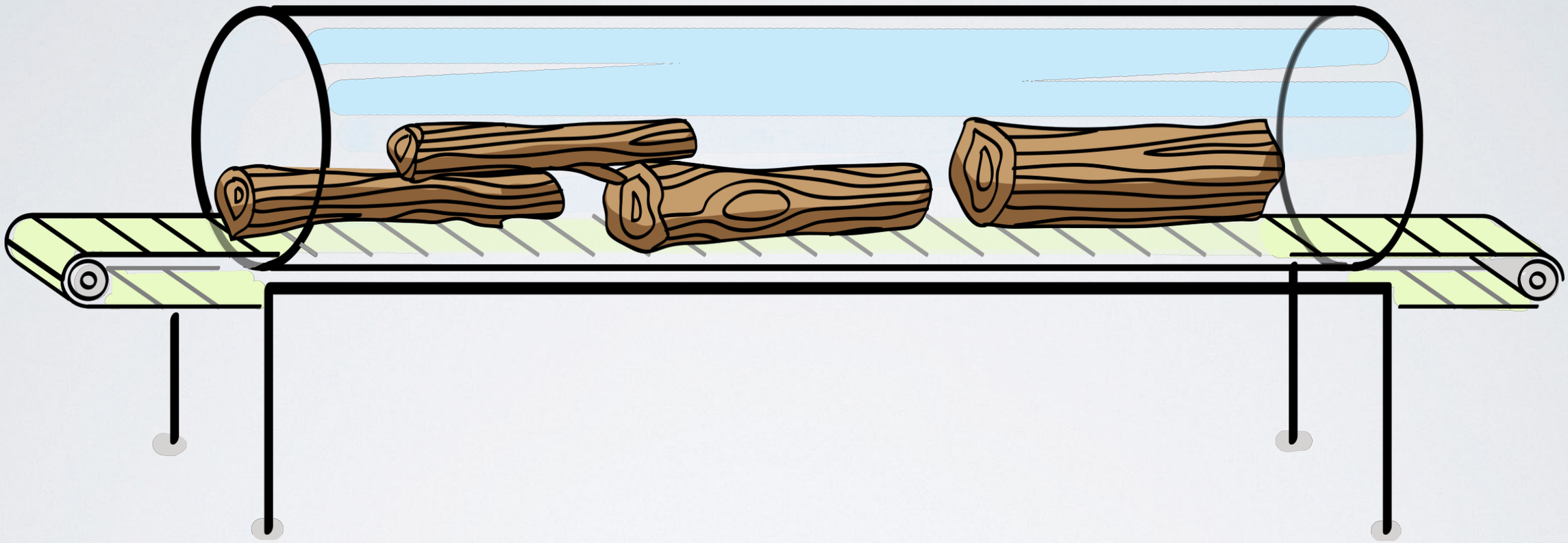
Recursion

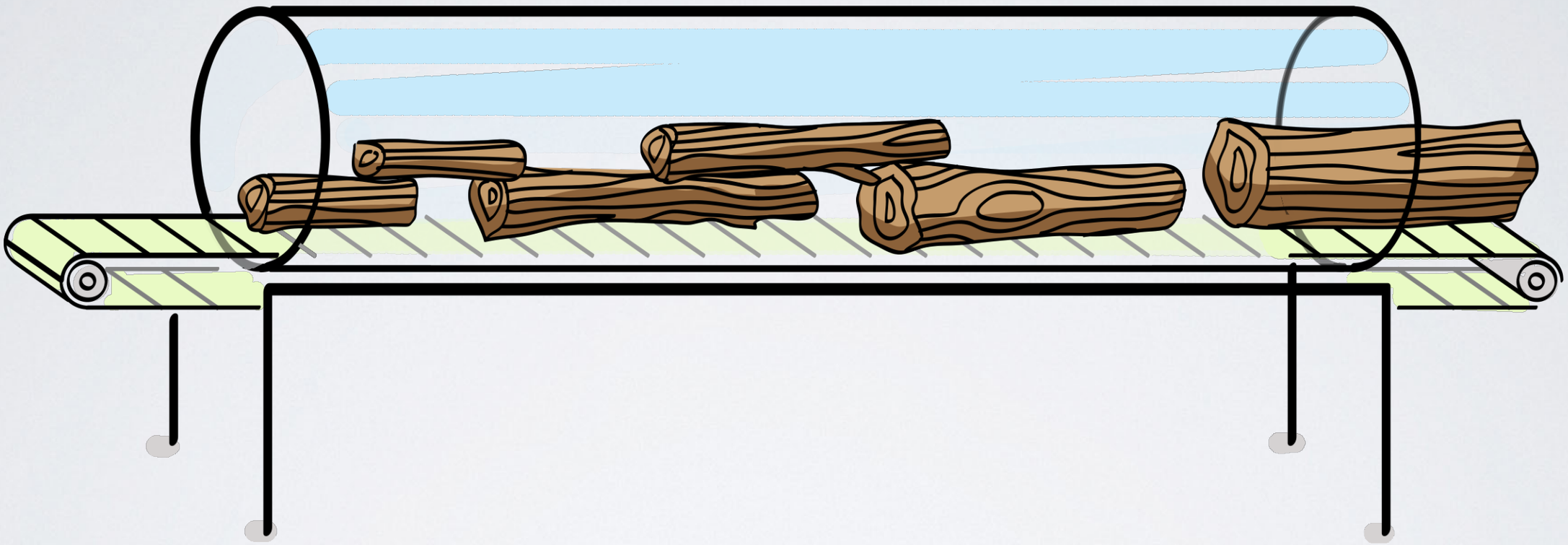


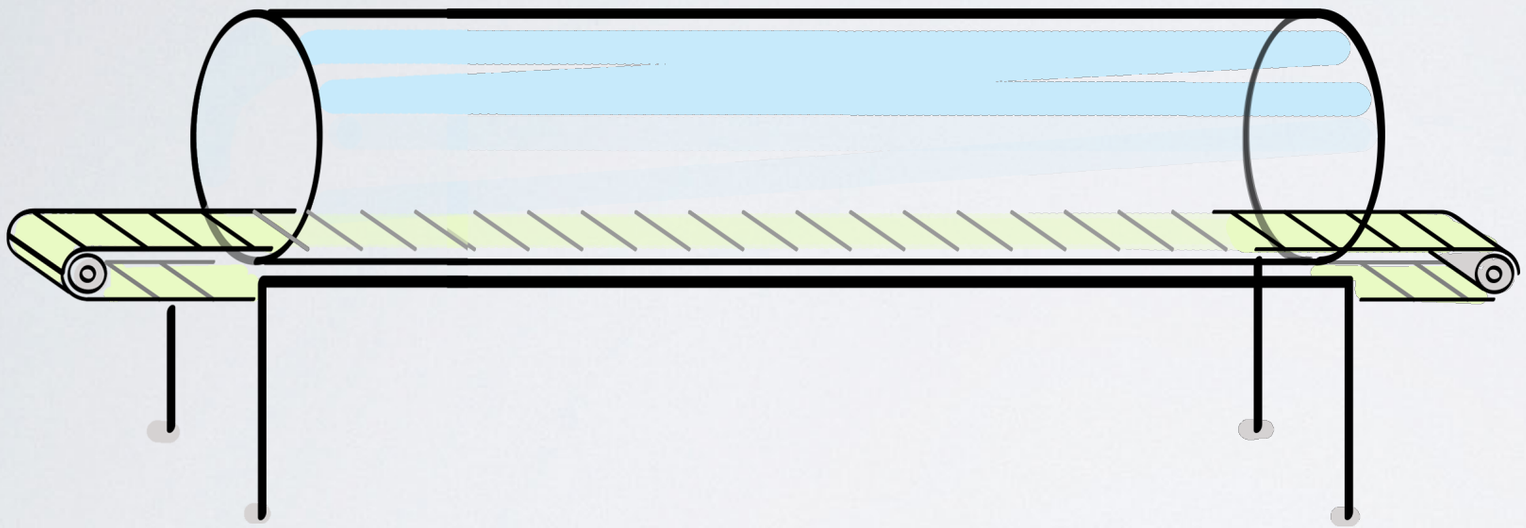


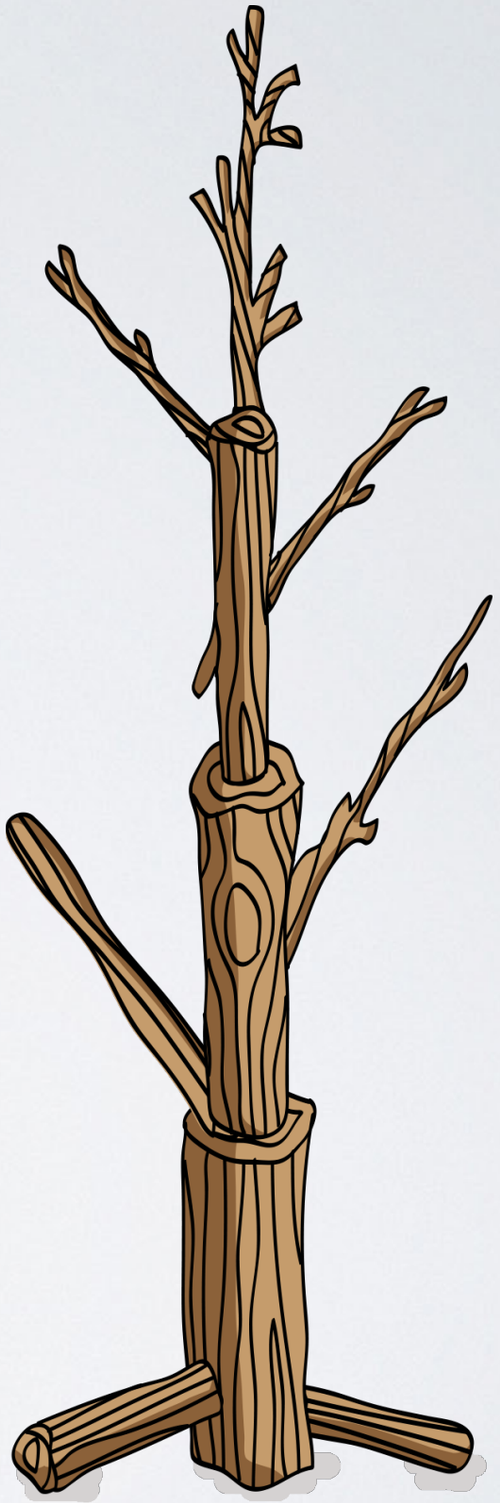












SESSION TYPES FOR LIST AND TREE SERVERS


```
type ListServer = &{
  Nil:  End,
  Cons: ?Int .
        ListServer
}
```

```
type TreeServer = &{
  LeafS:  Skip,
  NodeS: TreeServer ;
         ?Int ;
         TreeServer
}
```

TYPE EQUIVALENCE _ SOME LAWS

- Recursion is silently unfolded (equirecursive)

- Identity laws — T ; skip ~ T ; skip ~ T
- Associativity — (T ; U) ; V ~ T ; (U ; V)
- Distributivity — (&{l:T, m:U});V) ~ &{l:T;V, m:U;V}

LASIGE driven by excellence

FREEST

CONTEXT-FREE SESSION TYPES IN A CONCURRENT FUNCTIONAL LANGUAGE

LASIGE driven by excellence

DEMO AVAILABLE

```
data Tree = Leaf | Node Int Tree Tree

type TreeC = +{LeafC: Skip, NodeC: !Int;TreeC;TreeC;?Int}

transform : forall a => Tree -> TreeC;a -> (Tree, a)
transform tree c =
  case tree of {
  Leaf ->
    (Leaf, select LeafC c);
  Node x l r ->
    let c = select NodeC c in
    let c = send c x in
    let l, c = transform[TreeC;?Int;a] l c in
    let r, c = transform[?Int;a] r c in
    let y, c = receive c in
    (Node y l r, c)
}
```

```
treeSum : forall a => dualof TreeC;a -> (Int, a)
treeSum c =
  match c with {
    LeafC c -> (0, c);
    NodeC c ->
      let x, c = receive c in
      let l, c = treeSum[dualof TreeC;!Int;a] c in
      let r, c = treeSum[!Int;a] c in
      (x + l + r, send c (x + l + r))
  }

main : Tree
main =
  let w, r = new TreeC in
  let _ = fork (treeSum[Skip] r) in
  fst (transform[Skip] aTree w)
```

RAISING A TREE

```
data Tree =
  Leaf I
  Node Tree Int Tree

raiseTree c =
  case c of
    LeafS c0 → (Leaf, c0)
    NodeS c0 →
      let (t1, c1) = raiseTree c0
          (v, c2) = receive c1
          (t2, c3) = raiseTree c2
      in (Node t1 v t2, c3)
```

choice label

AIM: LOW-LEVEL SOLUTION

- Communication restricted to base types (no structures, no closures)
- First-order session types (no channel passing)

RAISING A LIST

```
type ListServer = &{
  Nil : end,
  Cons: ?Int . ListServer
}
```

:: ListServer

```
raiseList c =
  case c of
    Nil     → λc0. ([], c0)
    Cons → λc0.
      let (x, c1) = receive c0
          (xs, c2) = raiseList c1
      in (x:xs, c2)
```

branch on a choice type

read from a channel

WHAT TYPING SHOULD GUARANTEE

- No ill-formed streams

  Node 5 Leaf Leaf

  Tree Int Tree

- Stream does not end too soon

  Node Leaf 5

  missing right tree

- No extra elements after the end of the stream

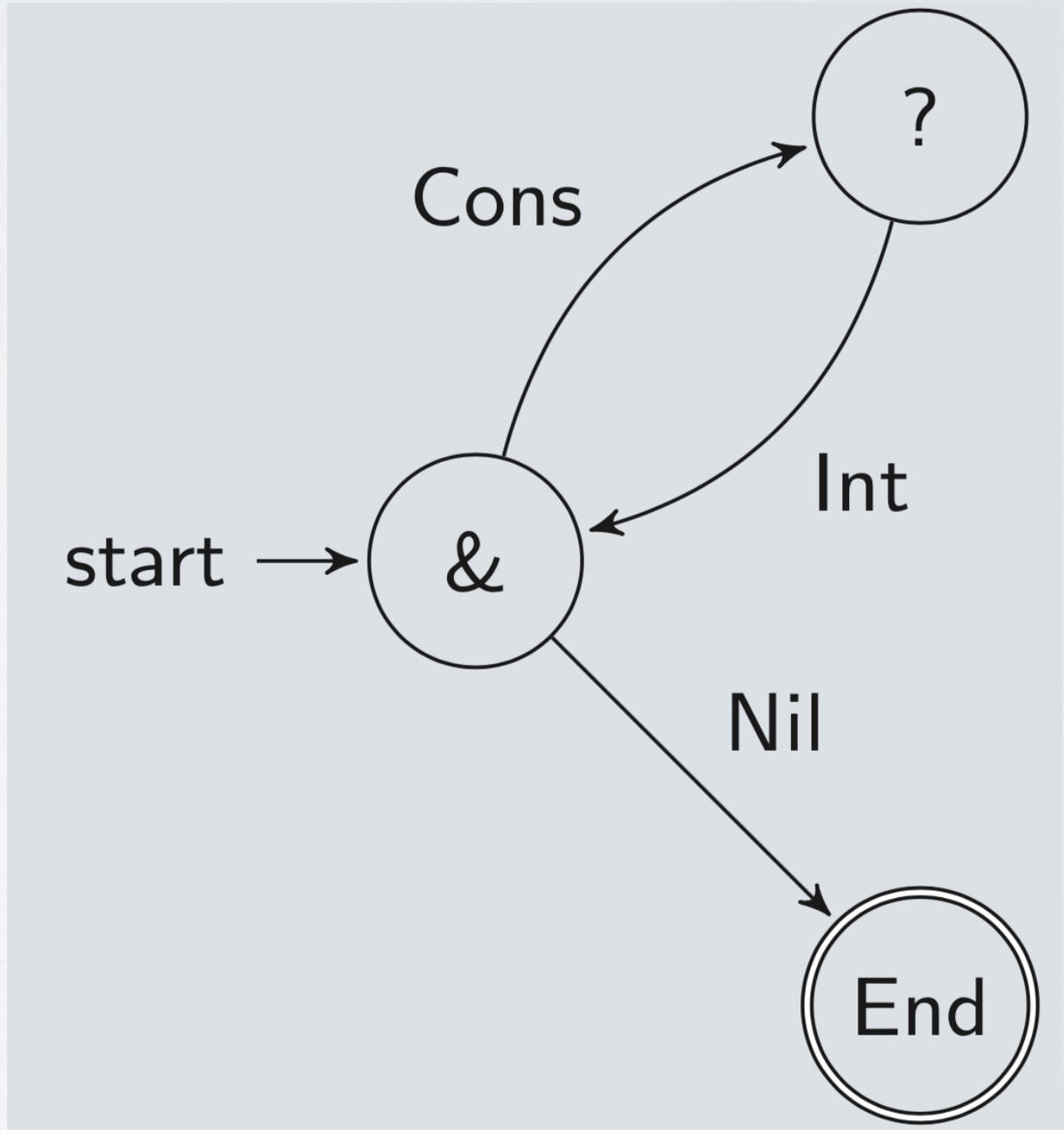
  Leaf 5

RECALL THE LIST SERVER

```
type ListServer = &{
  Nil : end,
  Cons: ?Int . ListServer
}
```

- Regular trace language

  (&Cons . ?Int)* . &Nil



THE TRACE LANGUAGE OF A SESSION TYPE

- The trace language of any (first-order) session type is:
  - recognised by a finite automaton
  - a (ω-) regular language
- The trace language of raiseTree is a context-free language:

$$T \rightarrow \&\mathsf{LeafS}$$
$$T \rightarrow \&\mathsf{NodeS}\ T\ ?\mathsf{Int}\ T$$

THEREFORE…

- There is no standard session type for raiseTree . It cannot be typed in (e.g.) GV

JFP **20** (1): 19–50, 2010. © Cambridge University Press 2009 19
doi:10.1017/S0956796809990268 First published online 8 December 2009

Linear type theory for asynchronous session types

SIMON J. GAY
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
(*e-mail:* simon@dcs.gla.ac.uk)

VASCO T. VASCONCELOS
Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal

LASIGE driven by excellence

ENTER CONTEXT-FREE SESSION TYPES

REGULAR & CONTEXT-FREE SESSION TYPES

Regular	Context free
!B.S	!B
?B.S	?B
end	S;S skip
Hardwired sequencing of communication	Explicit sequencing
Tail recursive	Unrestricted recursion

ARITHMETIC EXPRESSION SERVER


```
type TermChan = &{Const: !Int,
                  Add: TermChan;TermChan,
                  Mult: TermChan;TermChan}
computeService :: TermChan;!Int —o skip
computeService c =
  let n,c1 = receiveEval c in send n c1
receiveEval :: ∀α.TermChan;α —o Int ⊗ α
receiveEval c =
  case c of {
    Const c1 → receive c1
    Add c1 → let n1,c2 = receiveEval 1
                 n2,c3 = receiveEval c2
             in (n1+n2,c3)
    Mult c1 → let n1,c2 = receiveEval c …
}
```

LASIGE driven by excellence

REQUIREMENTS FOR THE META THEORY

TYPING RAISETREE

- At the top level:

```
type TreeServer = &{
  LeafS : skip,
  NodeS : TreeServer ;
          ?int ;
          TreeServer}
```

Linear function

Linear pair

raiseTree :: TreeServer —o (Tree ⊗ skip)

TYPING RAISETREE

- At the first recursive call

```
type TreeServer = &{
  LeafS : skip,
  NodeS : TreeServer ;
          ?int ;
          TreeServer}
```

```
raiseTree c = case c of
  Node → λc0. let (t1 , c1) = raiseTree c0
                  (v, c2) = receive c1
                  (t2 , c3) = raiseTree c2
```

raiseTree :: (TreeServer;?Int;TreeServer ;...) —o
Tree ⊗ (?Int; TreeServer;...)

TYPING RAISETREE

- At the top level:

raiseTree :: TreeServer —o (Tree ⊗ skip)

- At the first recursive call

raiseTree :: (TreeServer;?Int;TreeServer ;...) —o
Tree ⊗ (?Int; TreeServer;...)

- A polymorphic type

raiseTree :: ∀α. TreeServer;α —o (Tree ⊗ α)

INSTANTIATING A POLYMORPHIC TYPE

raiseTree :: ∀α. TreeServer;α —o (Tree ⊗ α)

- Replace α with skip

raiseTree :: TreeServer; skip —o (Tree ⊗ skip)

- We need

TreeServer; skip ~ TreeServer

TYPE EQUIVALENCE IS A BISIMULATION

- … over a suitable labelled transition system

- How do we decide type equivalence?

- For regular types there is a simple (polynomial) algorithm

- But context-free types is a different story…

TYPE EQUIVALENCE IS DECIDABLE

- Via translation to BPA, Basic Process Algebra [Bergstra, Klop 1988]

- Bisimulation for BPA is decidable [Christiansen, Hüttel, Stirling 1995]

- Unfortunately, the decidability result does not yield an obvious algorithm

CONCLUSION

- Programming with context free session types is fun
- Compiler ready soon
- Implementing type equivalence challenging