Repurposing Scala's **Pattern Matching** for Deeply Embedded **DSLs**

Tomas Mikula

Feb 19th, 2025

> scalac

Functional World

… in other words …

... in other words ...

Write DSL programs using **Scala**-embedded syntax

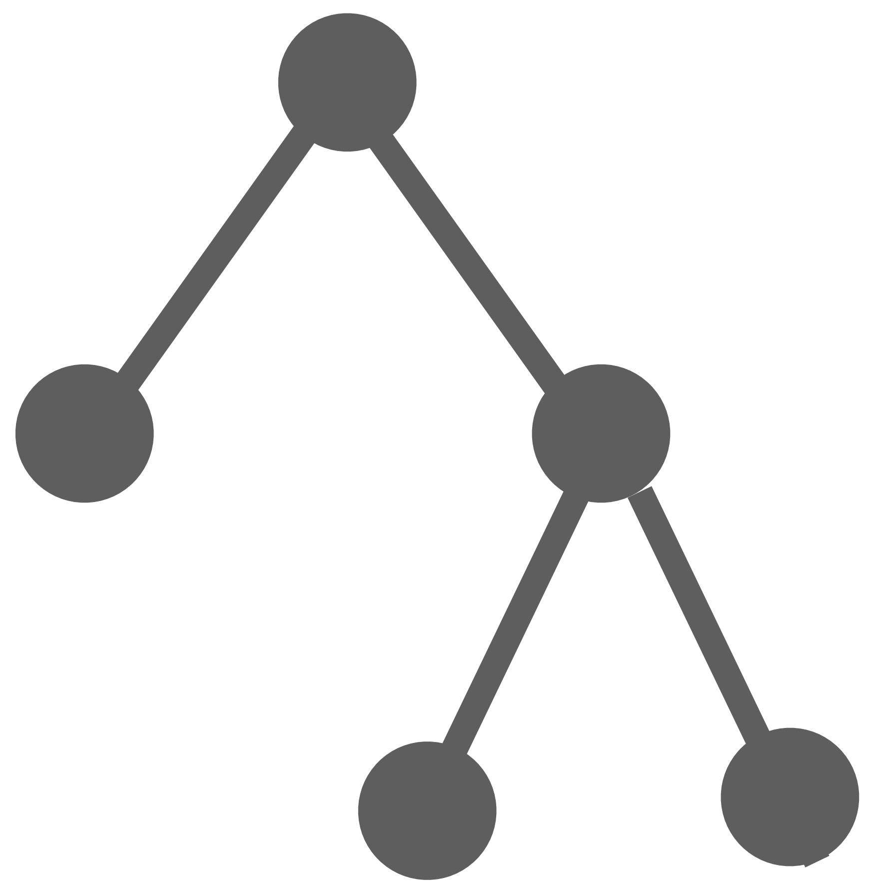
```
f(a) switch {
  case Foo(Bar(b) ** baz) =>
    doStuff(b ** baz)
  case Baz(c ** Qux(q)) =>
    doOtherStuff(c ** q)
}
```

… in other words …

Write DSL programs using
Scala-embedded syntax

```
f(a) switch {
  case Foo(Bar(b) ** baz) =>
    doStuff(b ** baz)
  case Baz(c ** Qux(q)) =>
    doOtherStuff(c ** q)
}
```

Get a reified representation
(your *custom* data structure)



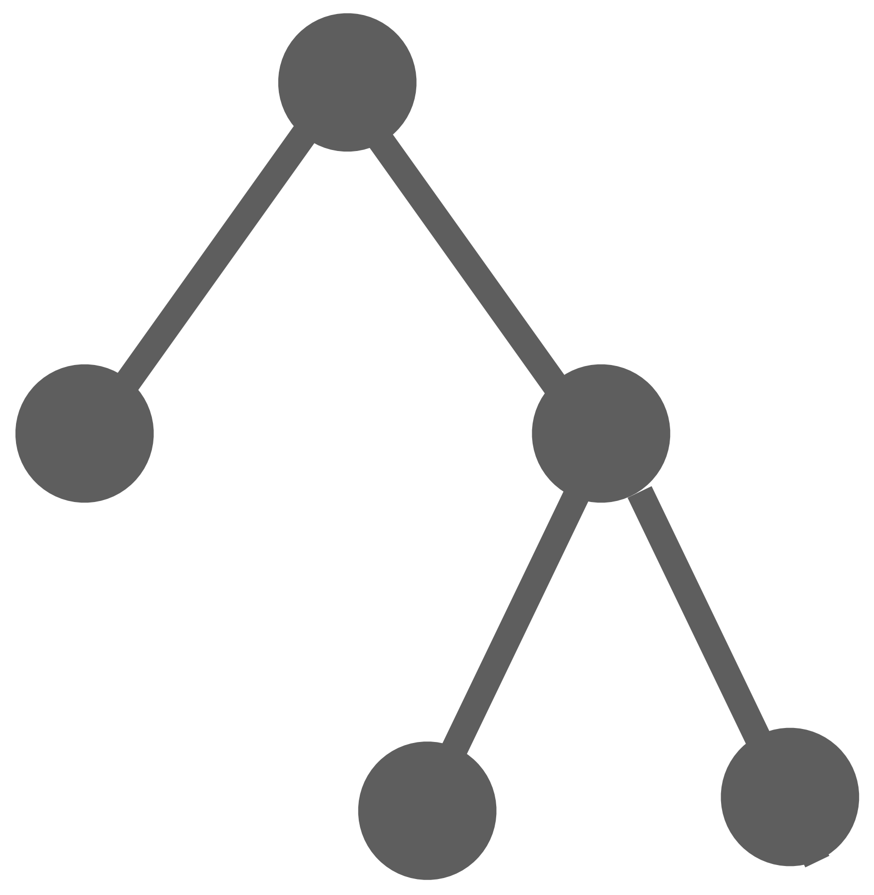
… in other words …

Write DSL programs using
Scala-embedded syntax

```
f(a) switch {
  case Foo(Bar(b) ** baz) =>
    doStuff(b ** baz)
  case Baz(c ** Qux(q)) =>
    doOtherStuff(c ** q)
}
```

When this *runs* …

Get a reified representation
(your *custom* data structure)



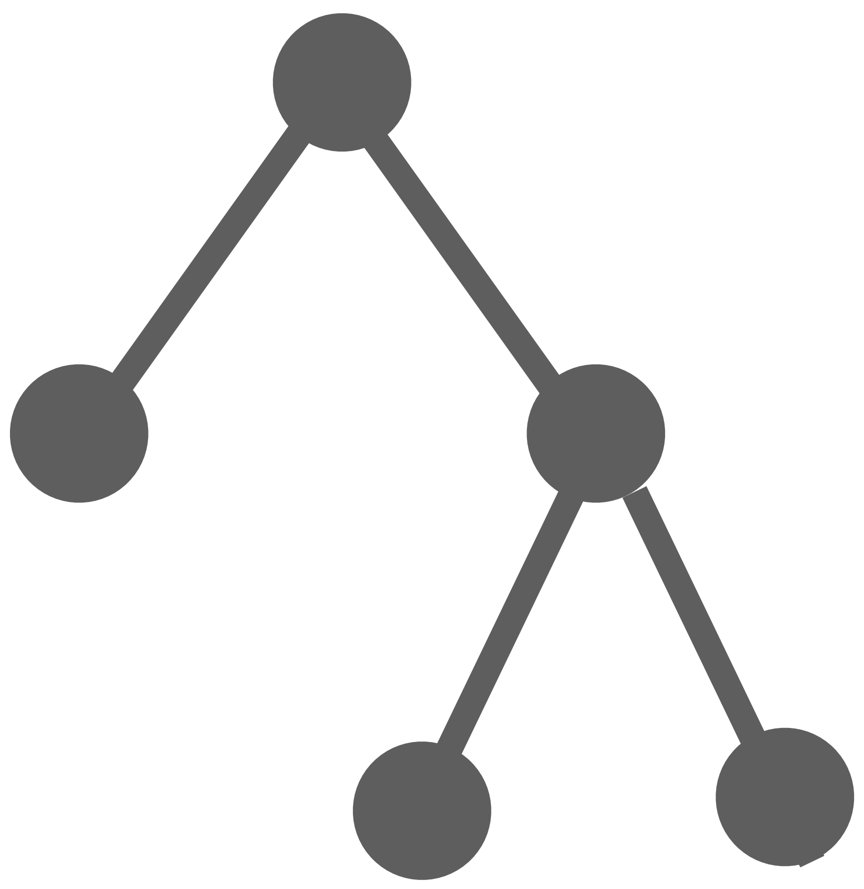
… in other words …

Write DSL programs using **Scala**-embedded syntax

```
f(a) switch {
  case Foo(Bar(b) ** baz) =>
    doStuff(b ** baz)
  case Baz(c ** Qux(q)) =>
    doOtherStuff(c ** q)
}
```

When this *runs* …

Get a reified representation (your *custom* data structure)



… it *constructs* this.

... in other words ...

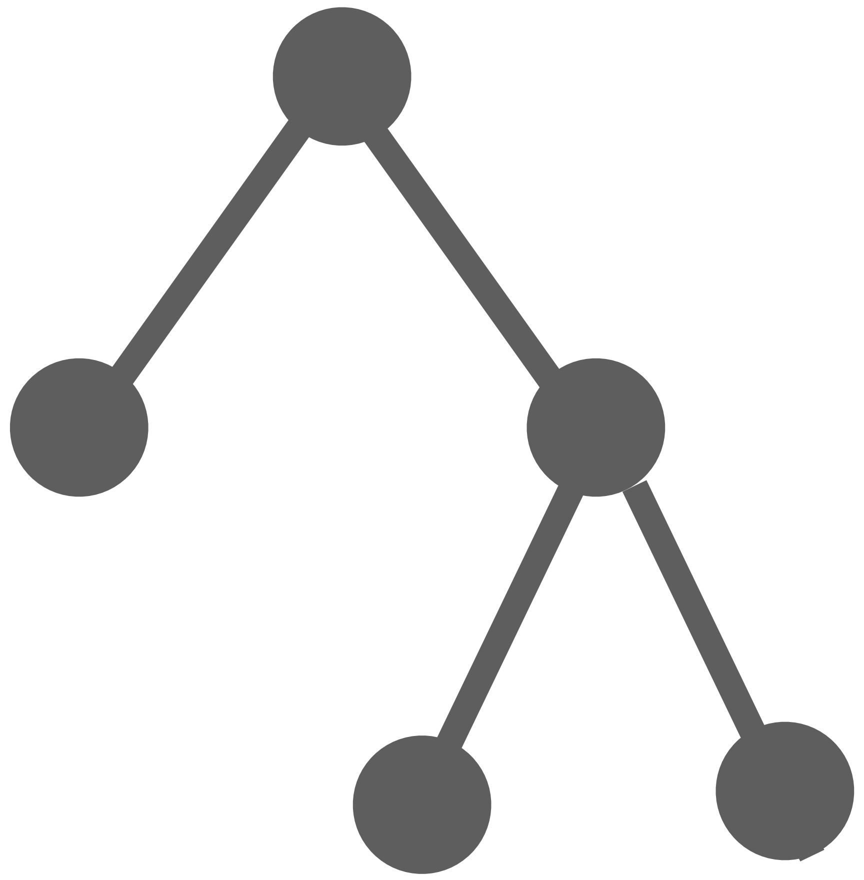
Write DSL programs using
Scala-embedded syntax

```
f(a) switch {
  case Foo(Bar(b) ** baz) =>
    doStuff(b ** baz)
  case Baz(c ** Qux(q)) =>
    doOtherStuff(c ** q)
}
```

When this *runs* …

(i.e. no macros)

Get a reified representation
(your *custom* data structure)



… it *constructs* this.

… in other words …

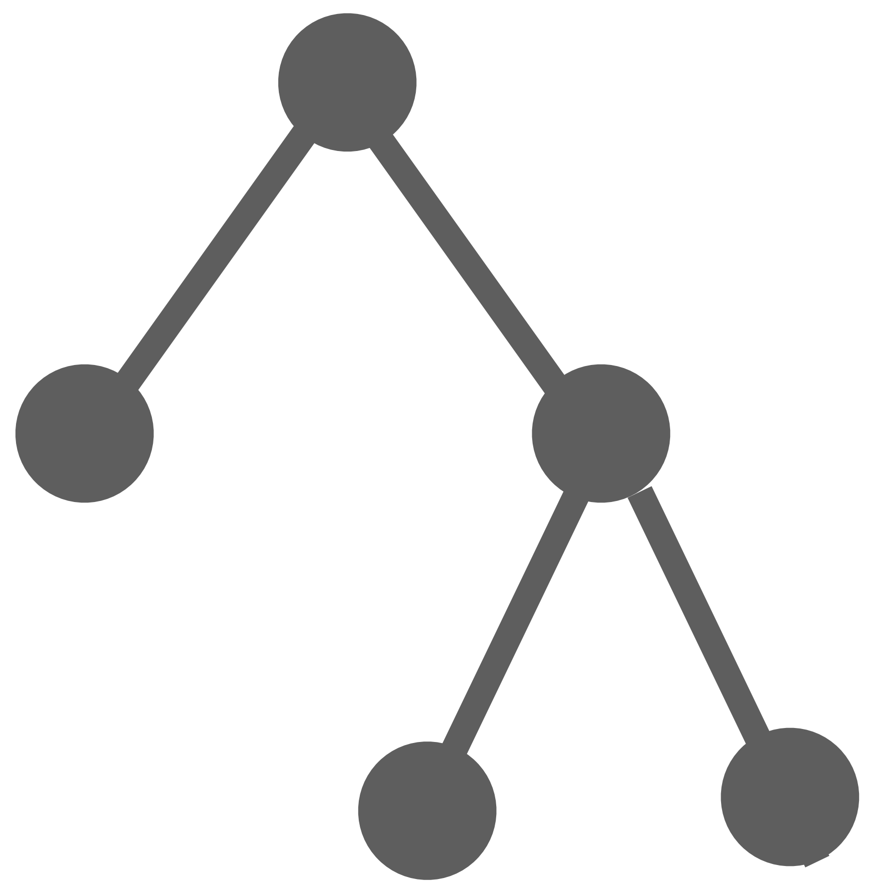
Write DSL programs using
Scala-embedded syntax

```
f(a) switch {
  case Foo(Bar(b) ** baz) =>
    doStuff(b ** baz)
  case Baz(c ** Qux(q)) =>
    doOtherStuff(c ** q)
}
```

When this *runs* …

(i.e. no macros)

Get a reified representation
(your *custom* data structure)



… it *constructs* this.

“DIY Scala Virtualized” (sort of)

Why, oh Why?

Why, oh Why?

Why **reified** intermediate representation?

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why bother **embedding** into Scala?

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why bother **embedding** into Scala?

Piggy-back on parser, type-checker, IDE.

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why bother **embedding** into Scala?

Piggy-back on parser, type-checker, IDE.

Why an uncanny **resemblance to Scala?**

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why bother **embedding** into Scala?

Piggy-back on parser, type-checker, IDE.

Why an uncanny **resemblance to Scala?**

Any better ideas for embedded pattern matching?

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why bother **embedding** into Scala?

Piggy-back on parser, type-checker, IDE.

Why an uncanny **resemblance to Scala?**

Any better ideas for embedded pattern matching?

Why would I need **pattern matching** *in* my DSL?

Why, oh Why?

Why **reified** intermediate representation?

Freedom of interpretation. (Visualization, Simulation, …)

Why bother **embedding** into Scala?

Piggy-back on parser, type-checker, IDE.

Why an uncanny **resemblance to Scala?**

Any better ideas for embedded pattern matching?

Why would I need **pattern matching** *in* my DSL?

Use case coming in a minute.

Aspects of a deeply-embedded DSL

Aspects of a deeply-embedded DSL

Domain

Aspects of a deeply-embedded DSL

Domain

Syntax

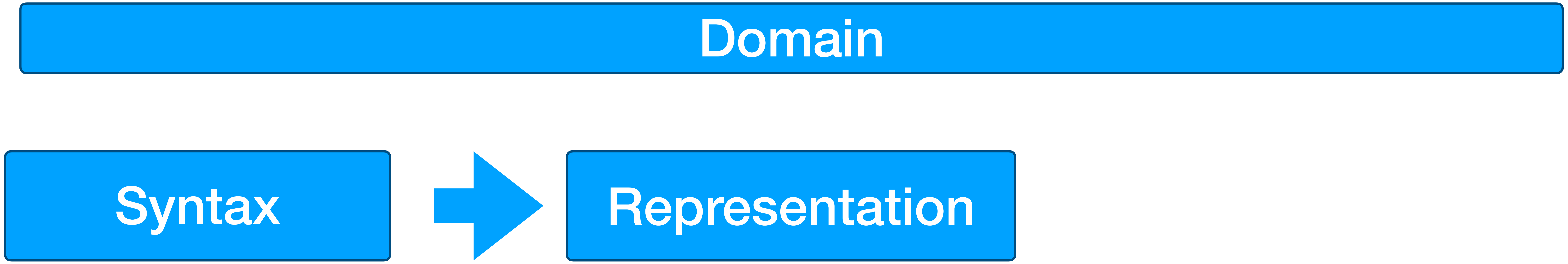
Aspects of a deeply-embedded DSL

Domain

Syntax

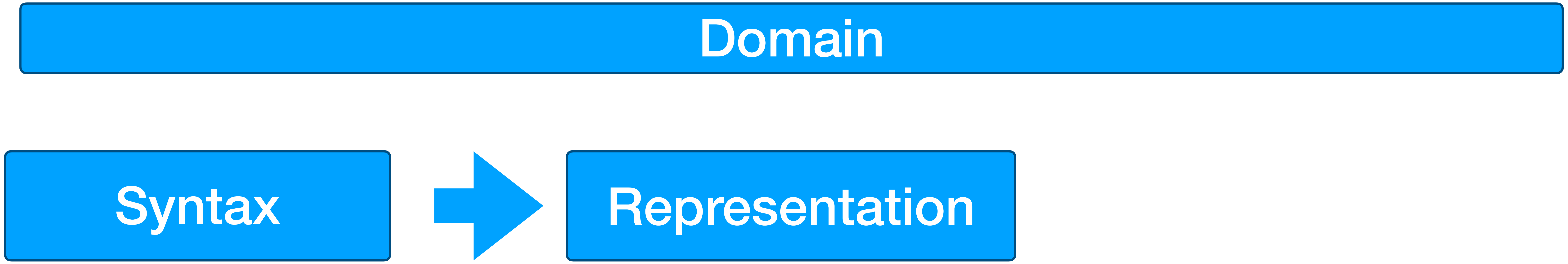
What we write

Aspects of a deeply-embedded DSL



What we write

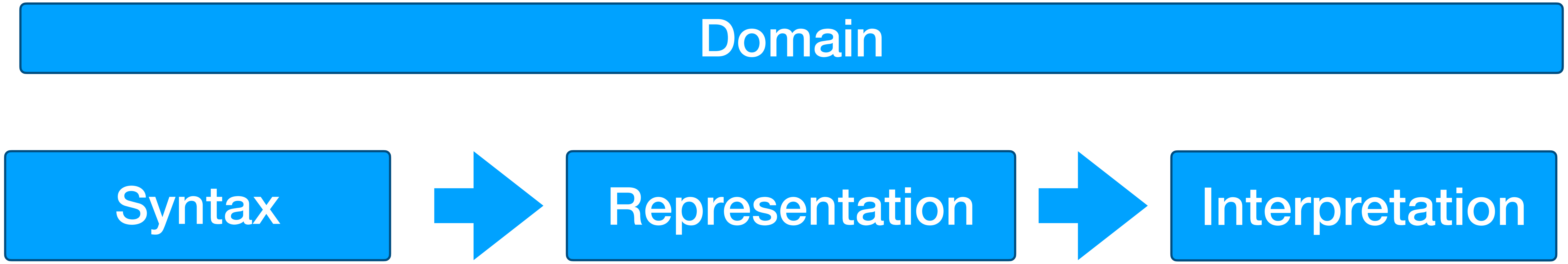
Aspects of a deeply-embedded DSL



What we write

What we prefer to operate on

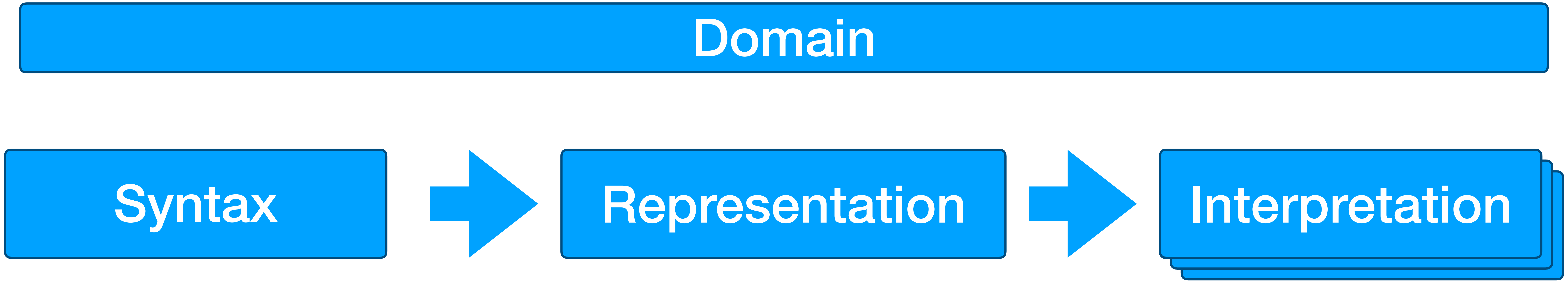
Aspects of a deeply-embedded DSL



What we write

What we prefer to operate on

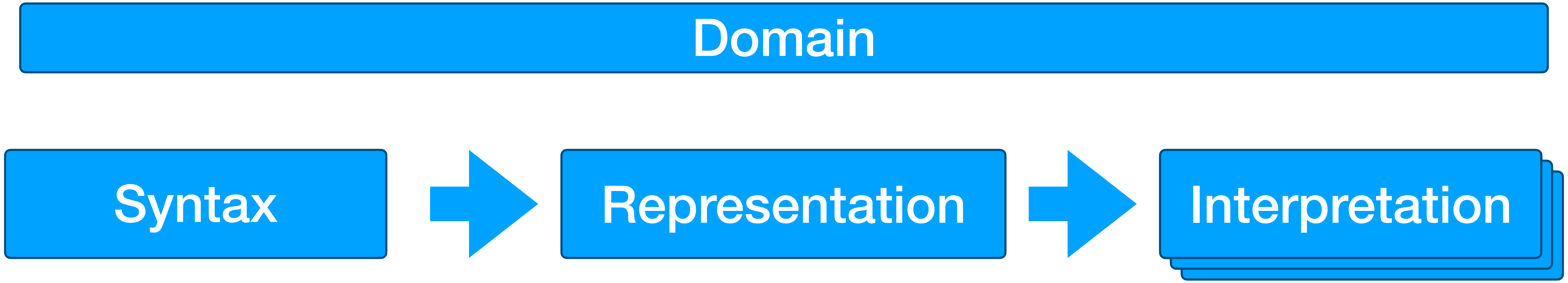
Aspects of a deeply-embedded DSL



What we write

What we prefer to operate on

Aspects of a deeply-embedded DSL

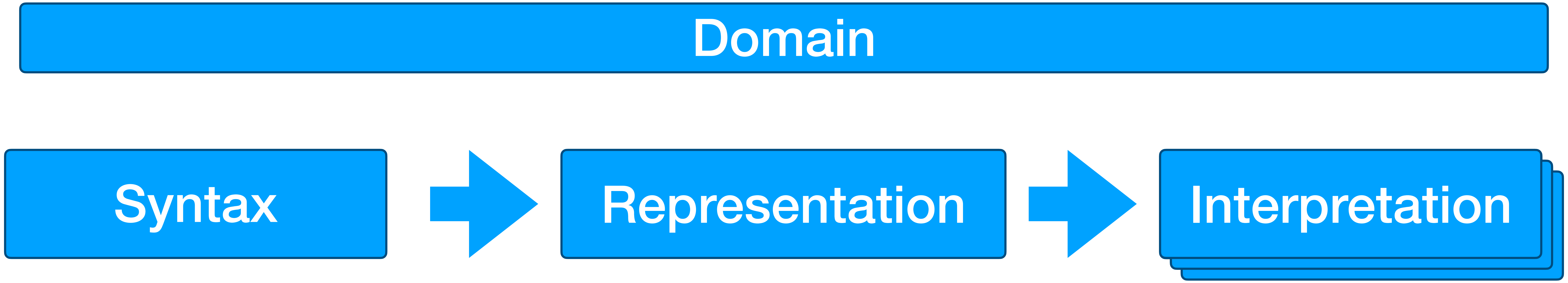


What we write

What we prefer to operate on

- Run

Aspects of a deeply-embedded DSL

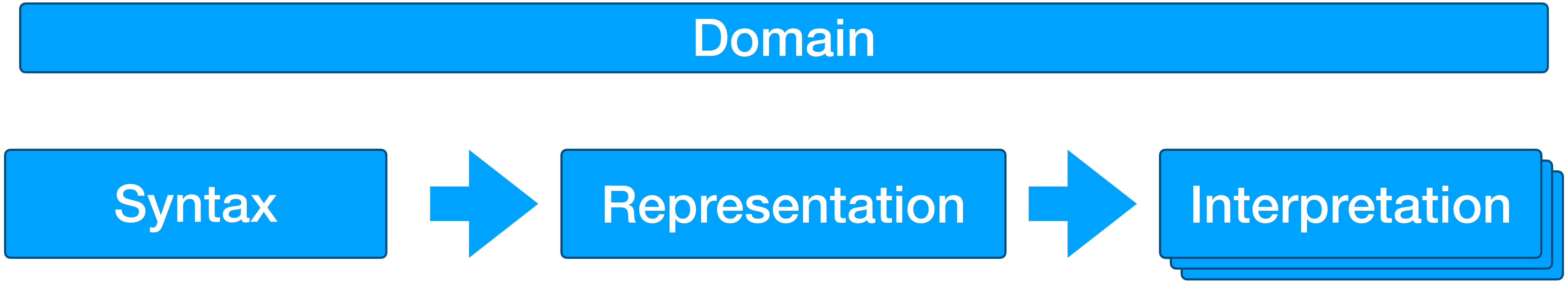


What we write

What we prefer to operate on

- Run
- Visualize

Aspects of a deeply-embedded DSL

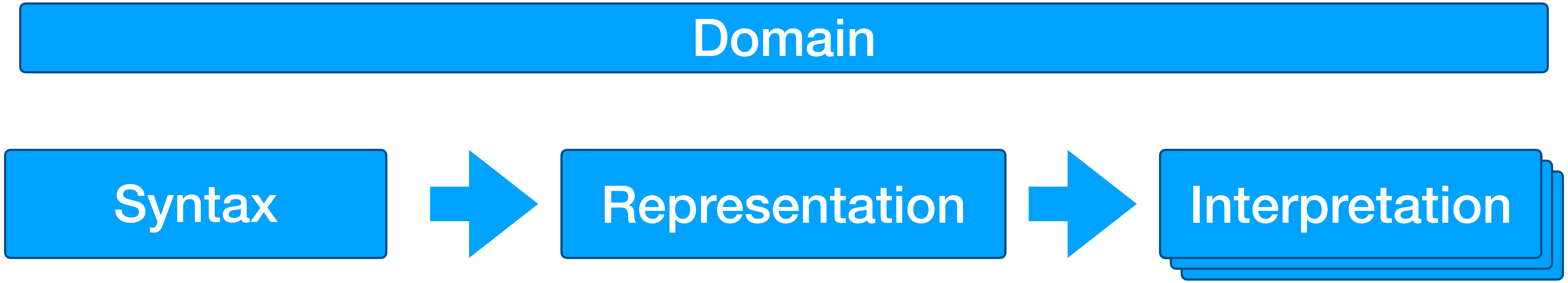


What we write

What we prefer to operate on

- Run
- Visualize
- Serialize

Aspects of a deeply-embedded DSL

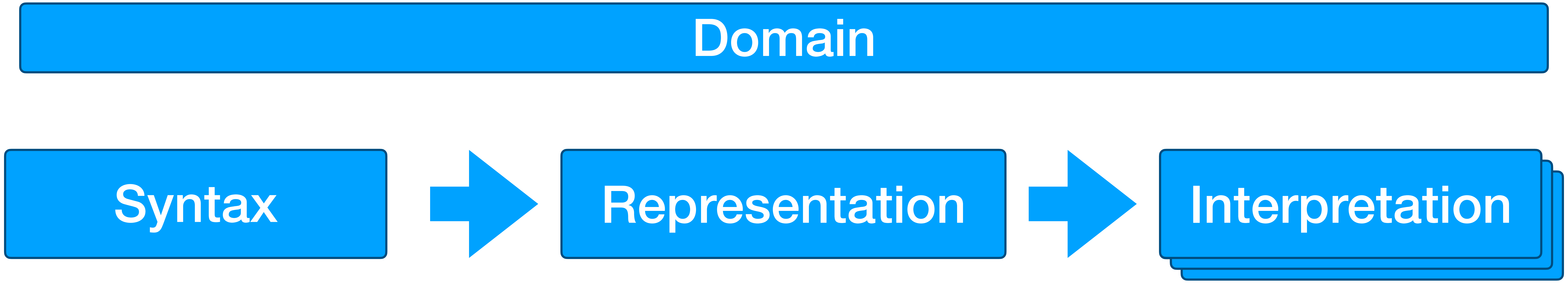


What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile

Aspects of a deeply-embedded DSL

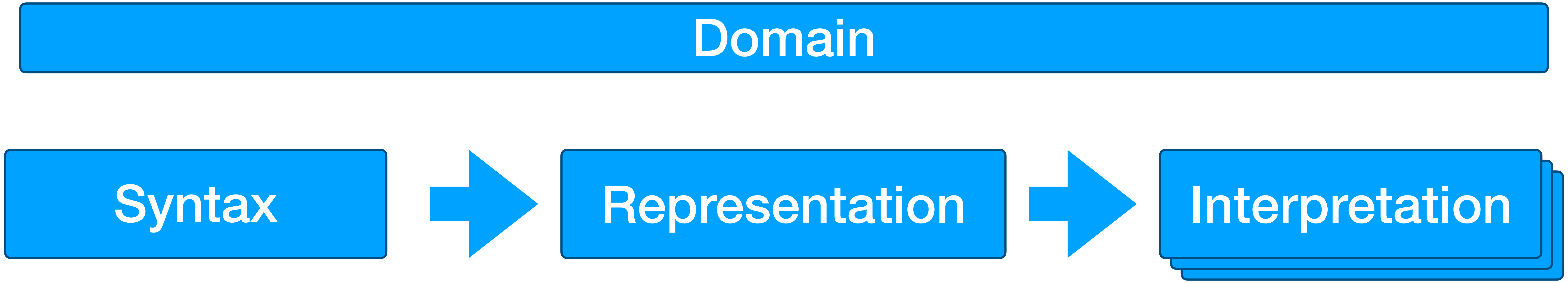


What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile

Aspects of a deeply-embedded DSL



What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate

Aspects of a deeply-embedded DSL

Domain

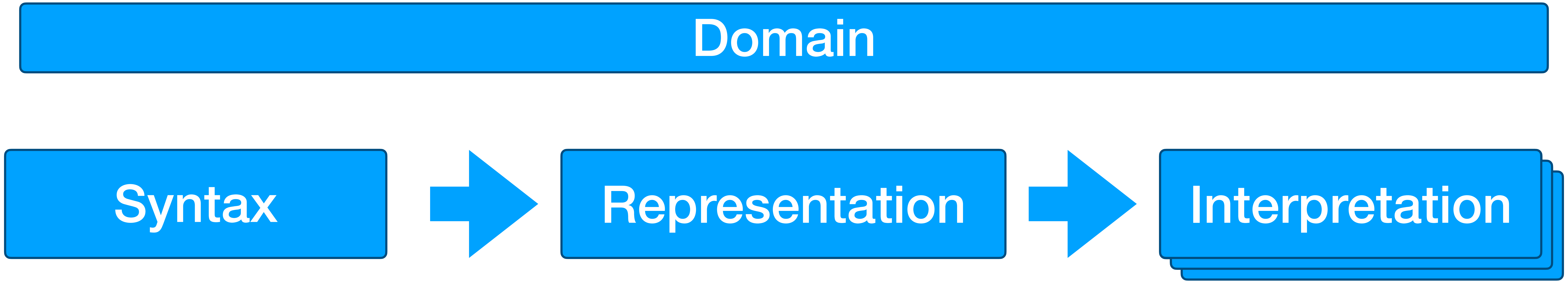
Syntax → Representation → Interpretation

What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate
- …

Aspects of a deeply-embedded DSL



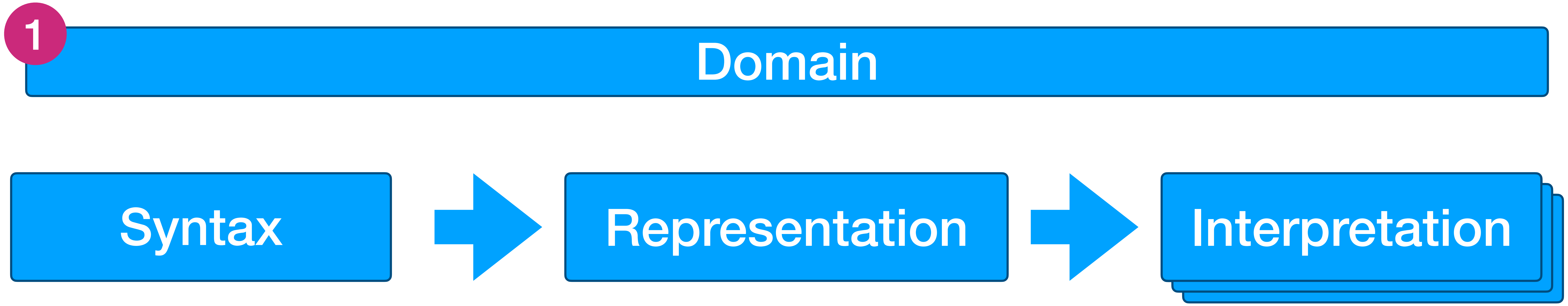
What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate
- …

Agenda:

Aspects of a deeply-embedded DSL



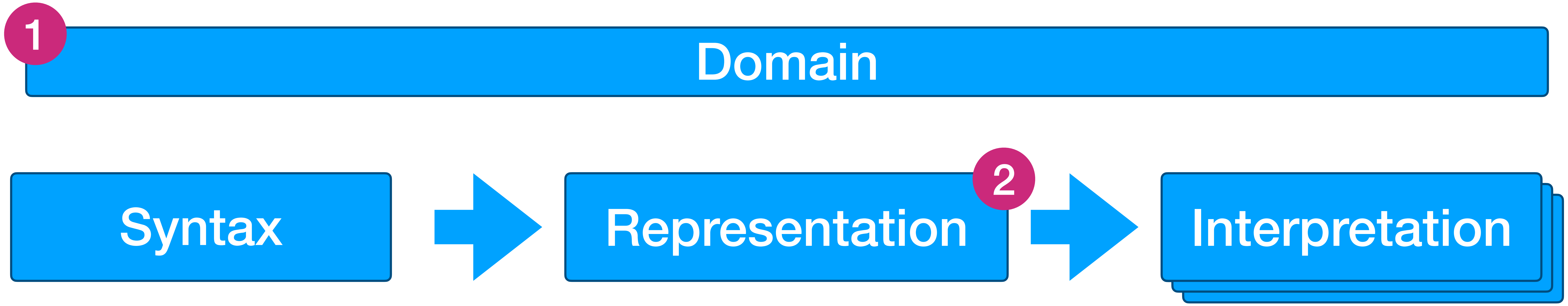
What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate
- …

Agenda: 1

Aspects of a deeply-embedded DSL



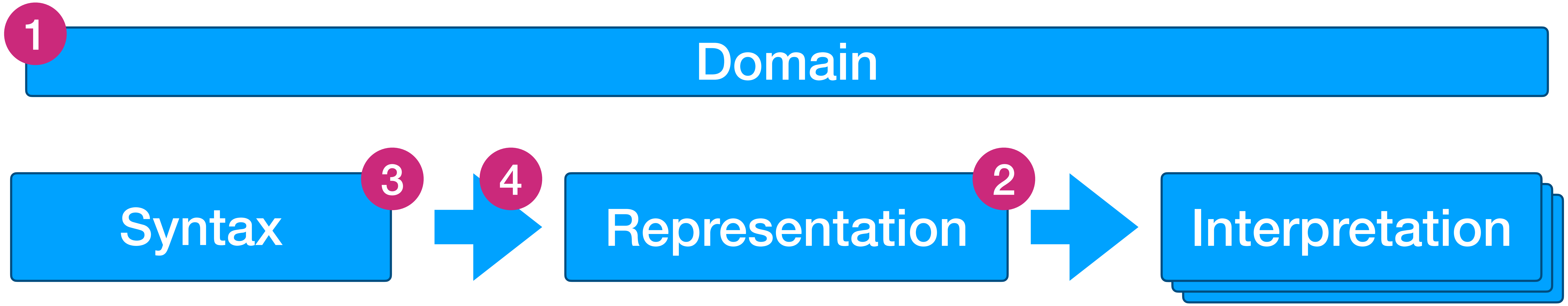
What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate
- …

Agenda: 1 2

Aspects of a deeply-embedded DSL



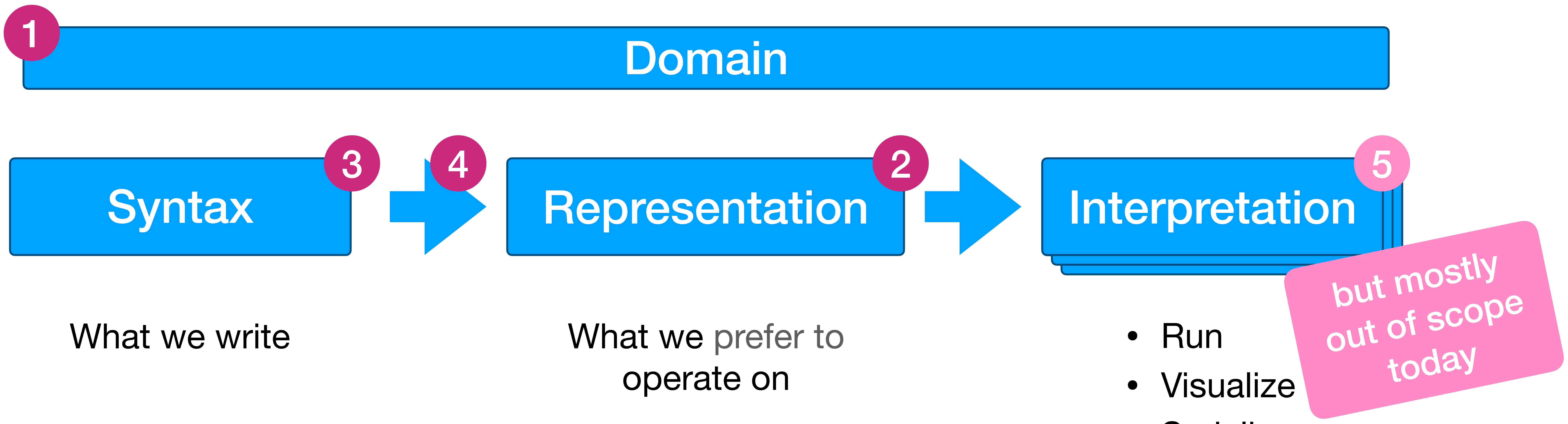
What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate
- …

Agenda: 1 2 3 4

Aspects of a deeply-embedded DSL



What we write

What we prefer to operate on

- Run
- Visualize
- Serialize
- Compile
- Transpile
- Simulate
- …

Agenda: 1 2 3 4 5

Domain: Workflows

Domain: **Workflows**

“A fault-oblivious stateful function that orchestrates activities.”

— *cadenceworkflow.io*

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

Domain: **Workflows**

“A fault-oblivious stateful function that orchestrates activities.”

— *cadenceworkflow.io*

activity

- a **business-level** operation

Domain: **Workflows**

“A fault-oblivious stateful function that orchestrates activities.”

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

- **business logic** on top of activities

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

- **business logic** on top of activities
- potentially long-running

Domain: Workflows

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

- **business logic** on top of activities
- potentially long-running
- **resilient** w.r.t.

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

- **business logic** on top of activities
- potentially long-running
- **resilient** w.r.t.
  - infrastructure failures

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

- **business logic** on top of activities
- potentially long-running
- **resilient** w.r.t.
  - infrastructure failures
  - intermittent activity failures

Domain: **Workflows**

"A fault-oblivious stateful function that orchestrates activities."

— *cadenceworkflow.io*

activity

- a **business-level** operation
- e.g. calling a service
- possibly **effectful**
- possibly **non-deterministic**

workflow

- **business logic** on top of activities
- potentially long-running
- **resilient** w.r.t.
  - infrastructure failures
  - intermittent activity failures
  - **without explicit database**

Example: Equipment Request

Example: Equipment Request

- Request work equipment

Example: Equipment Request

- Request work equipment
    - monitor, chair

Example: Equipment Request

- Request work equipment
  - monitor, chair
- For use

Example: Equipment Request

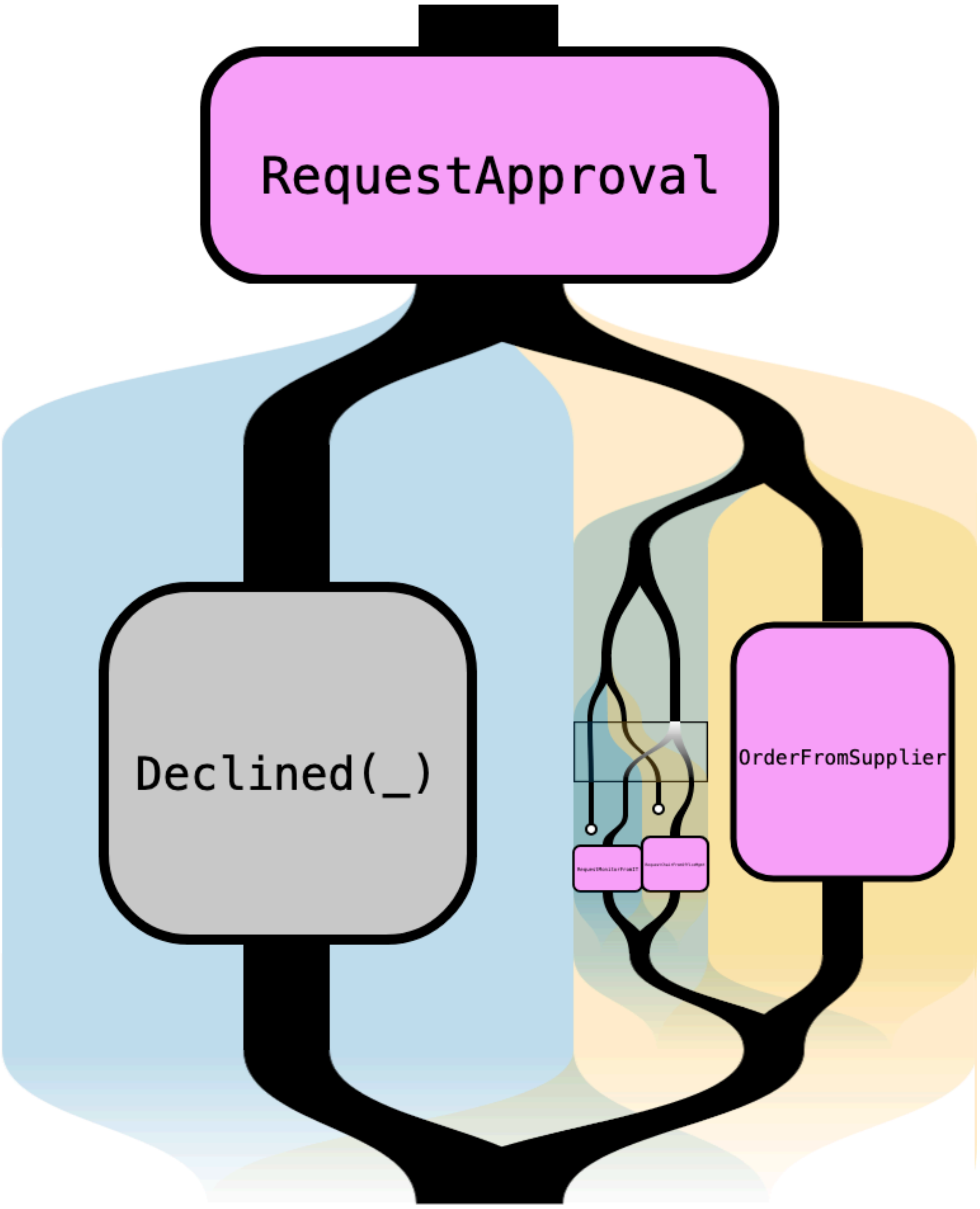
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)

Example: Equipment Request

- Request work equipment
    - monitor, chair
- For use
    - In the office (+ desk location)
    - at home (+ delivery address)

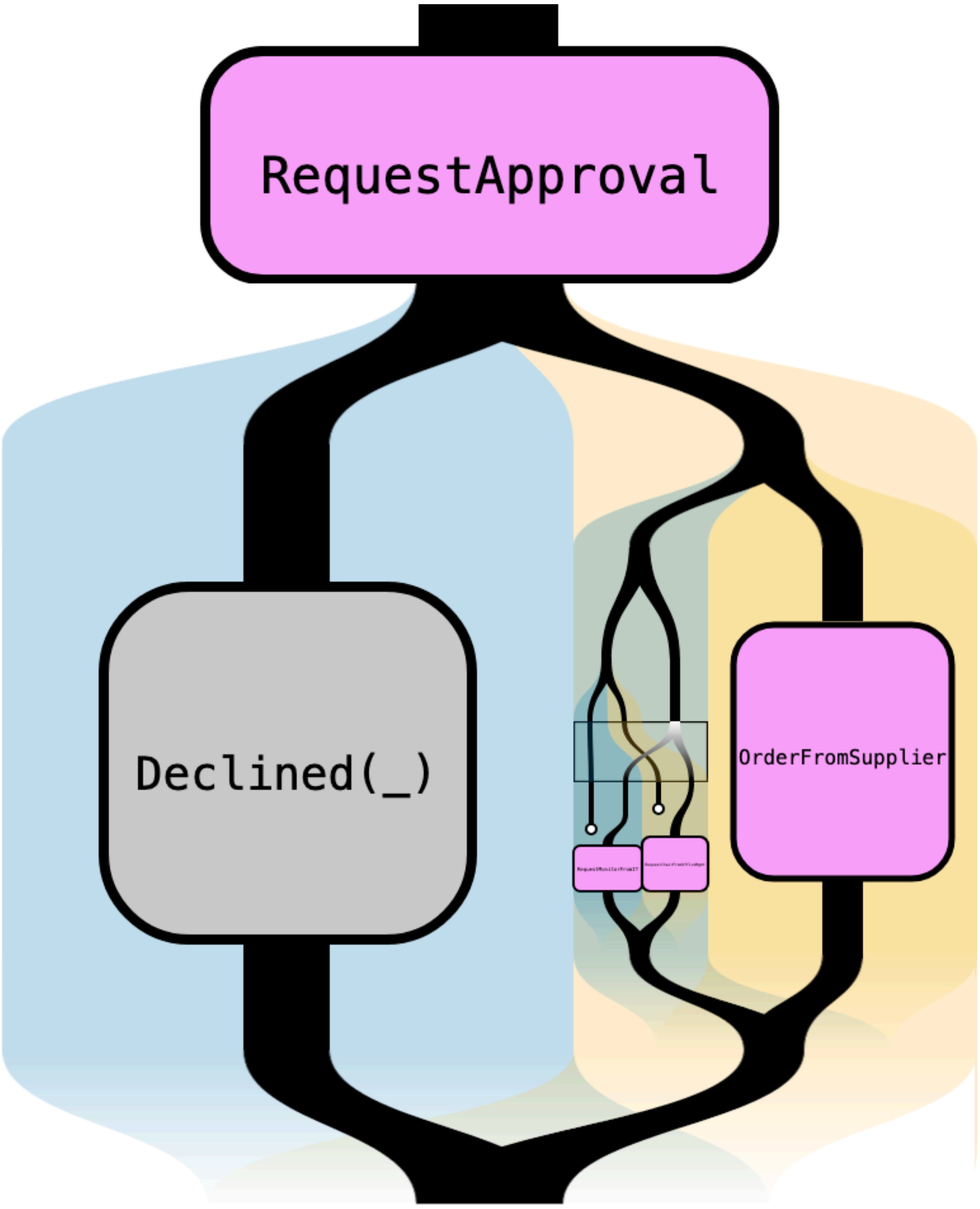
Example: Equipment Request

- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



Example: Equipment Request

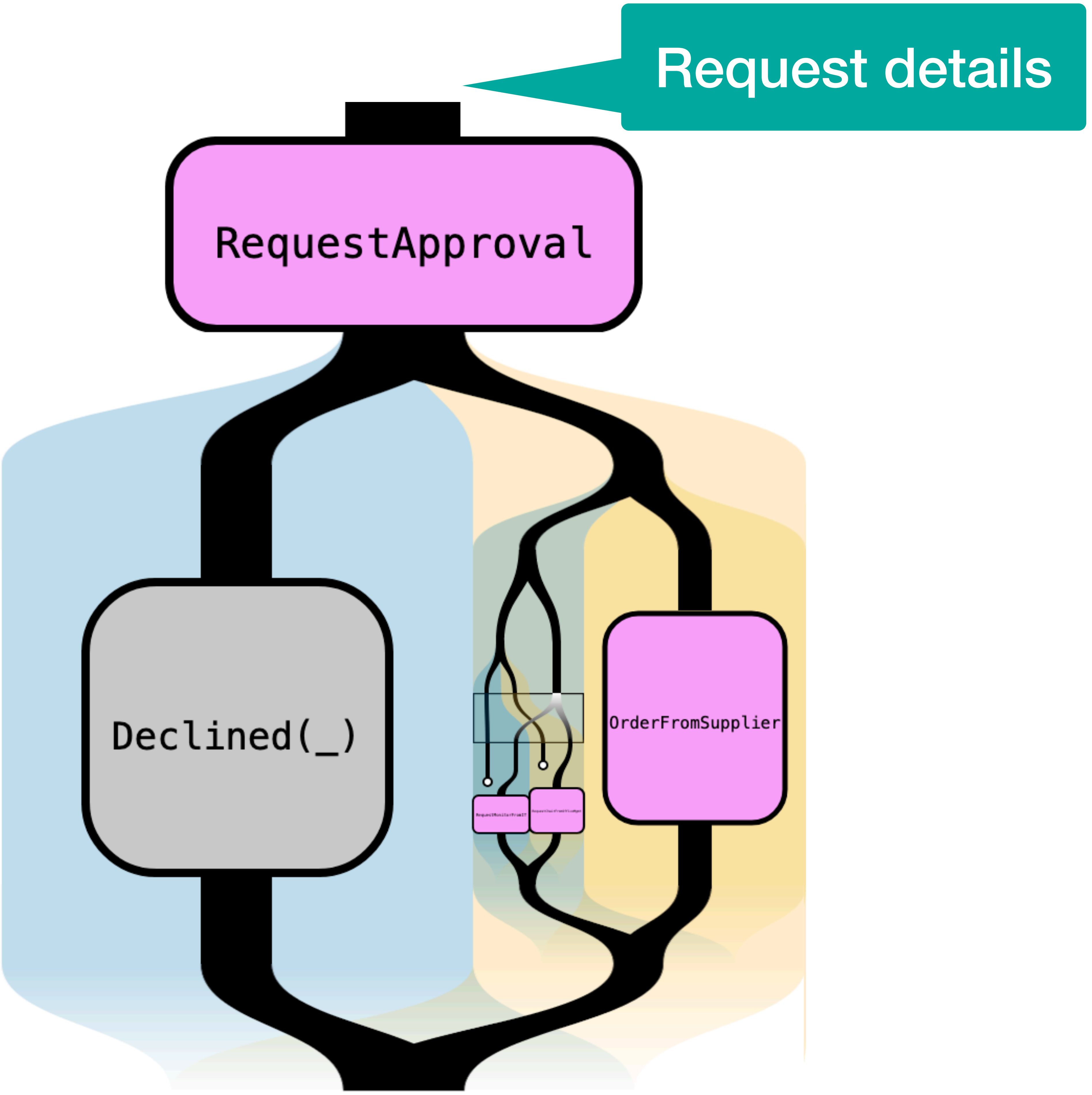
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)

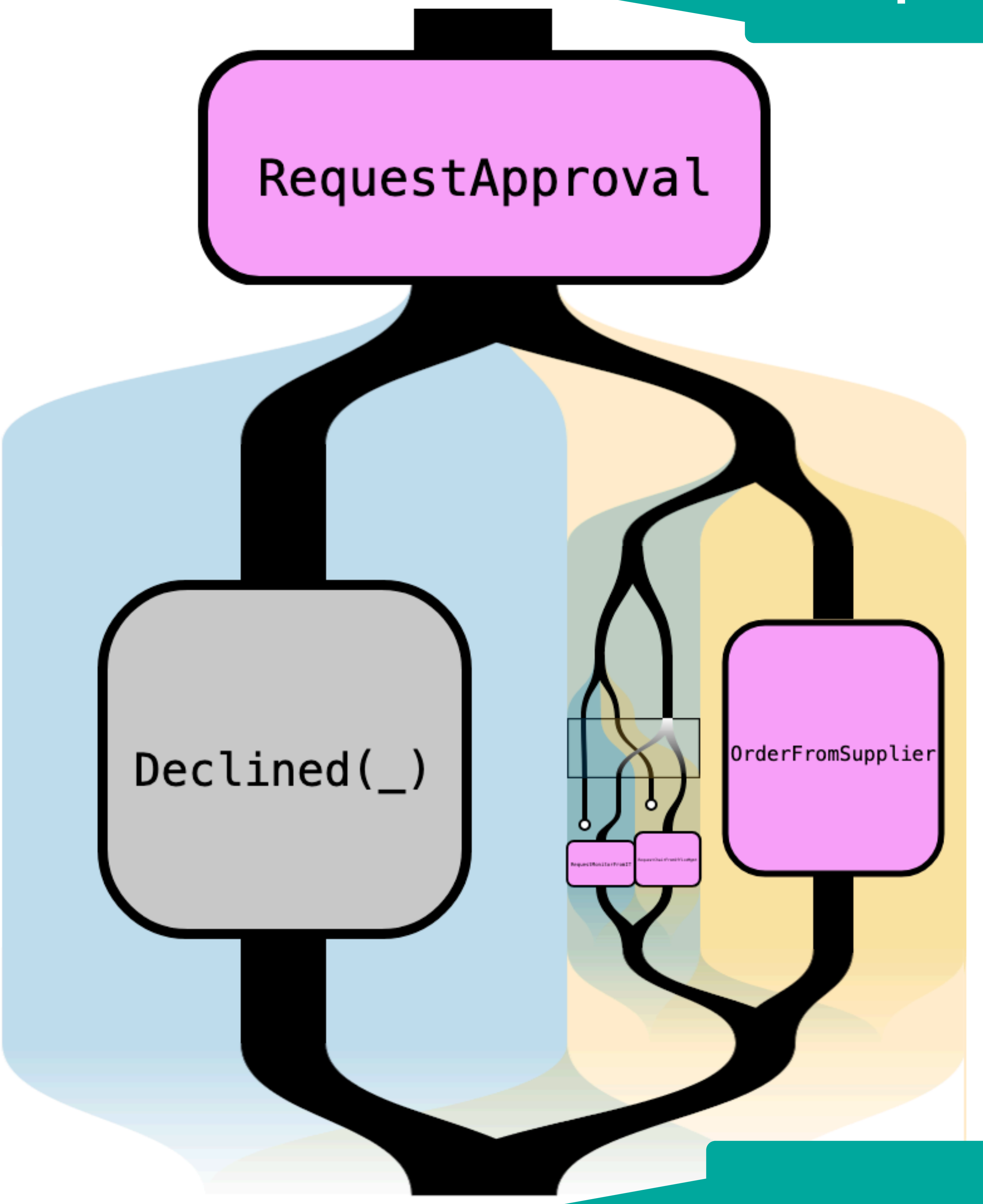


(mostly) generated from code

Example: Equipment Request

- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)

Request details

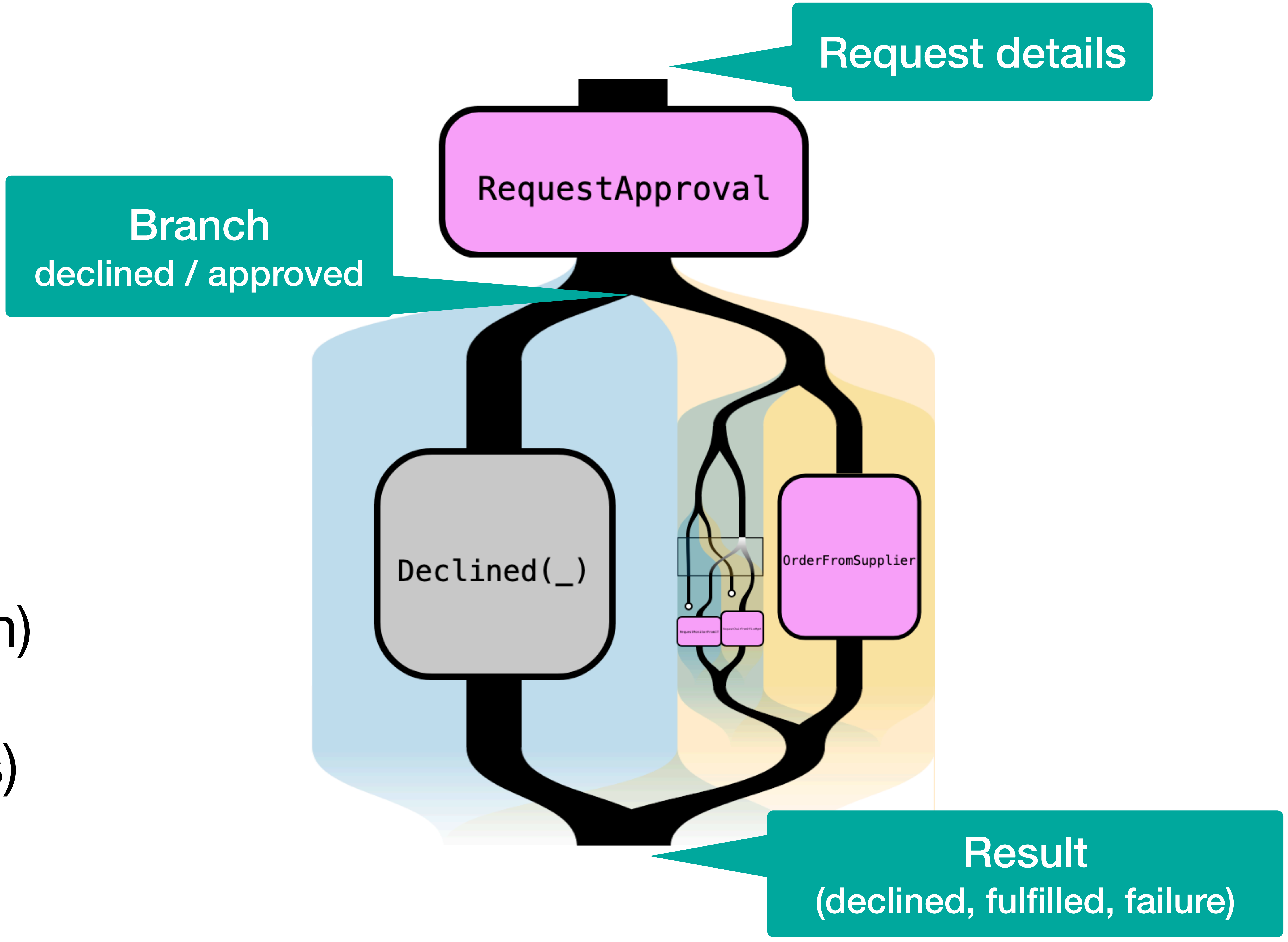


Result
(declined, fulfilled, failure)

(mostly) generated from code

Example: Equipment Request

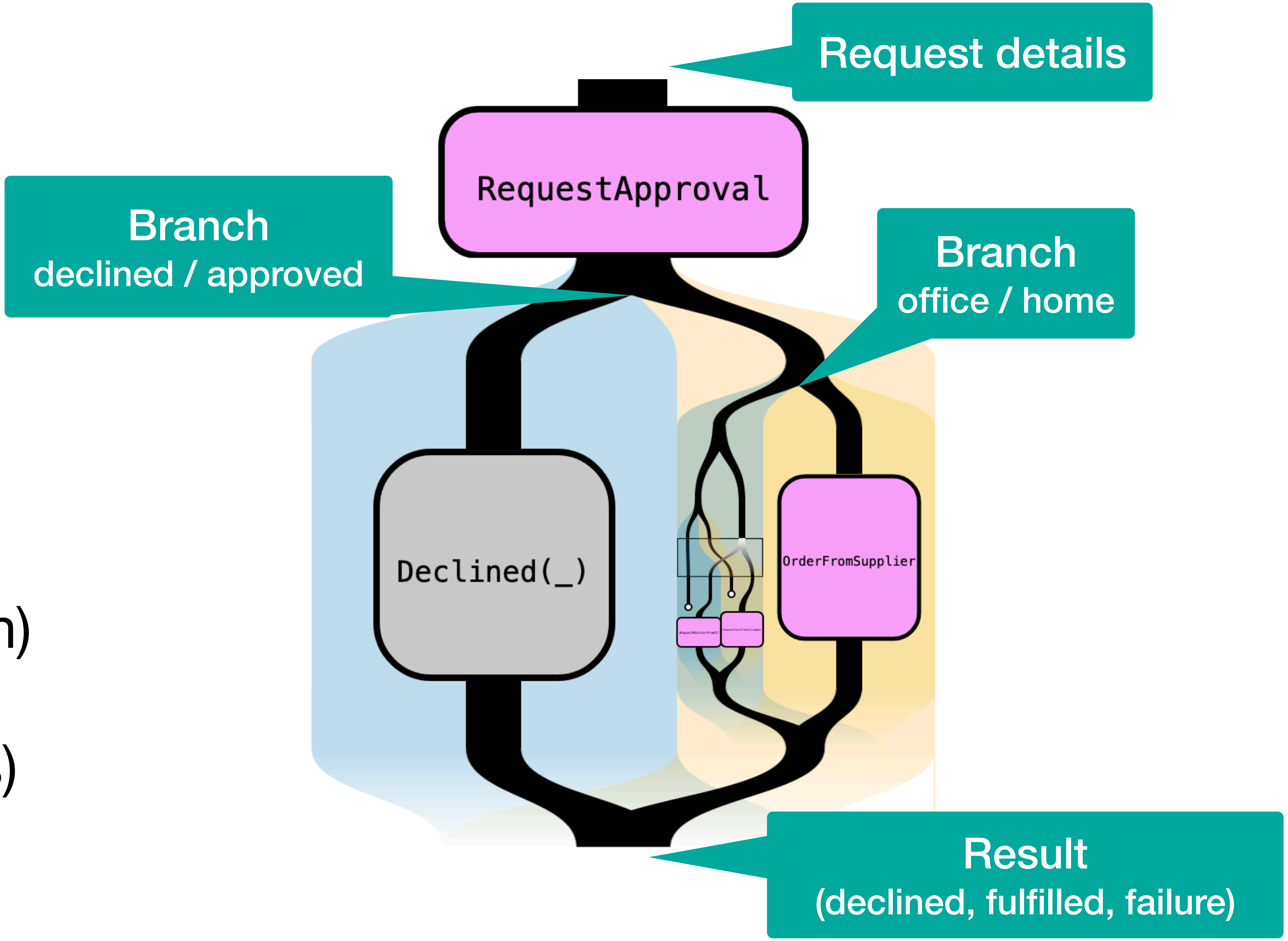
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

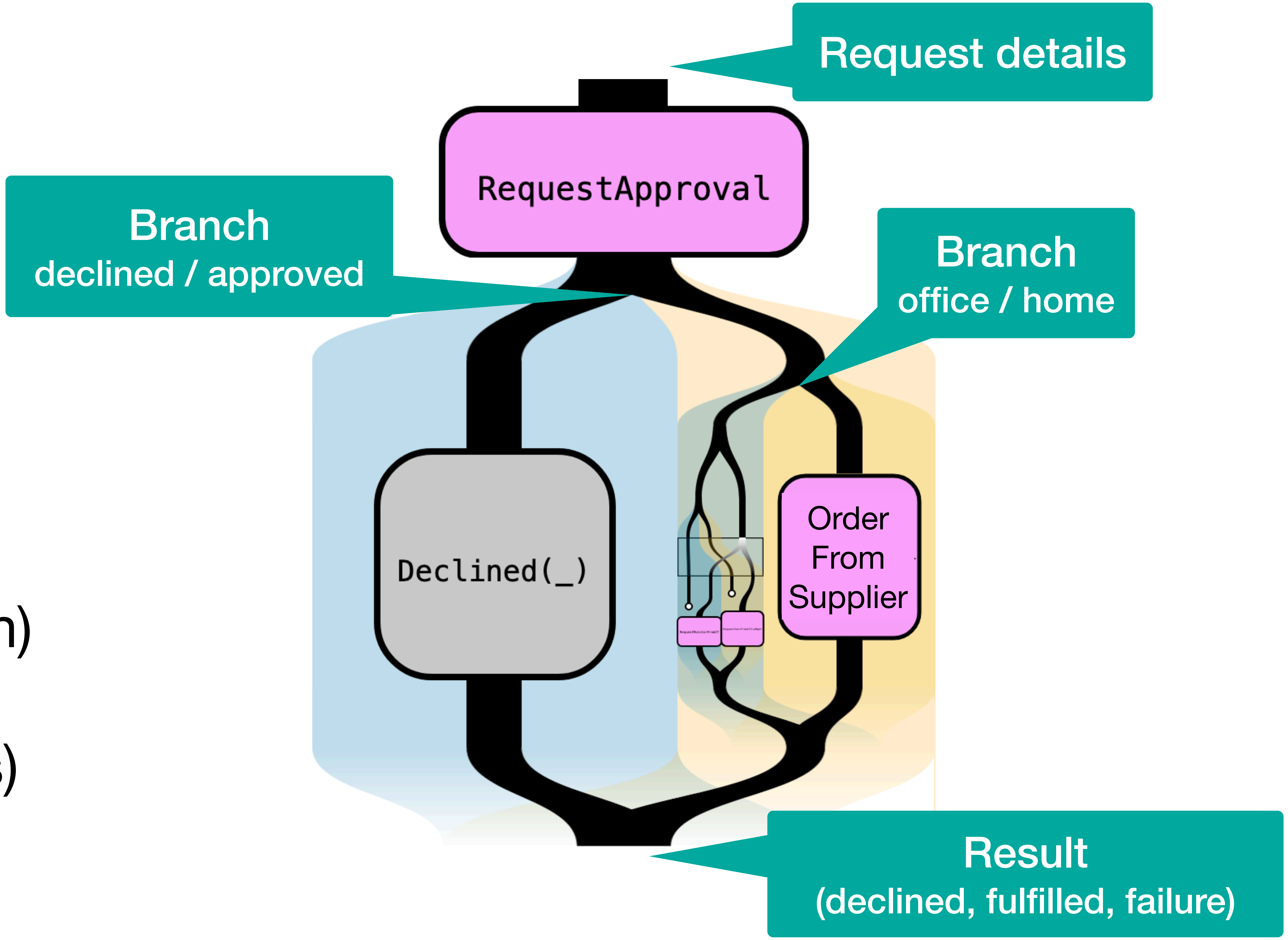
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

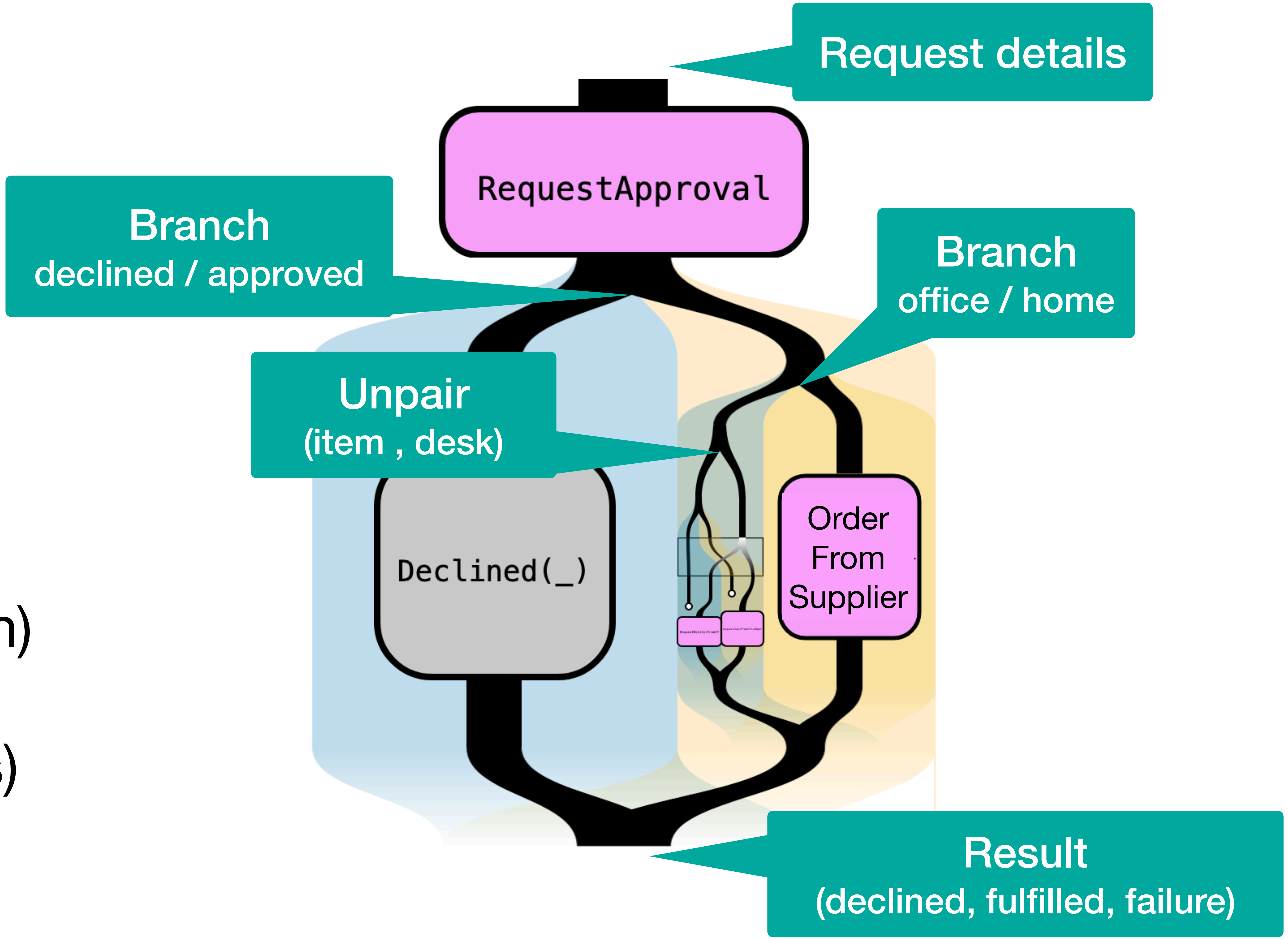
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

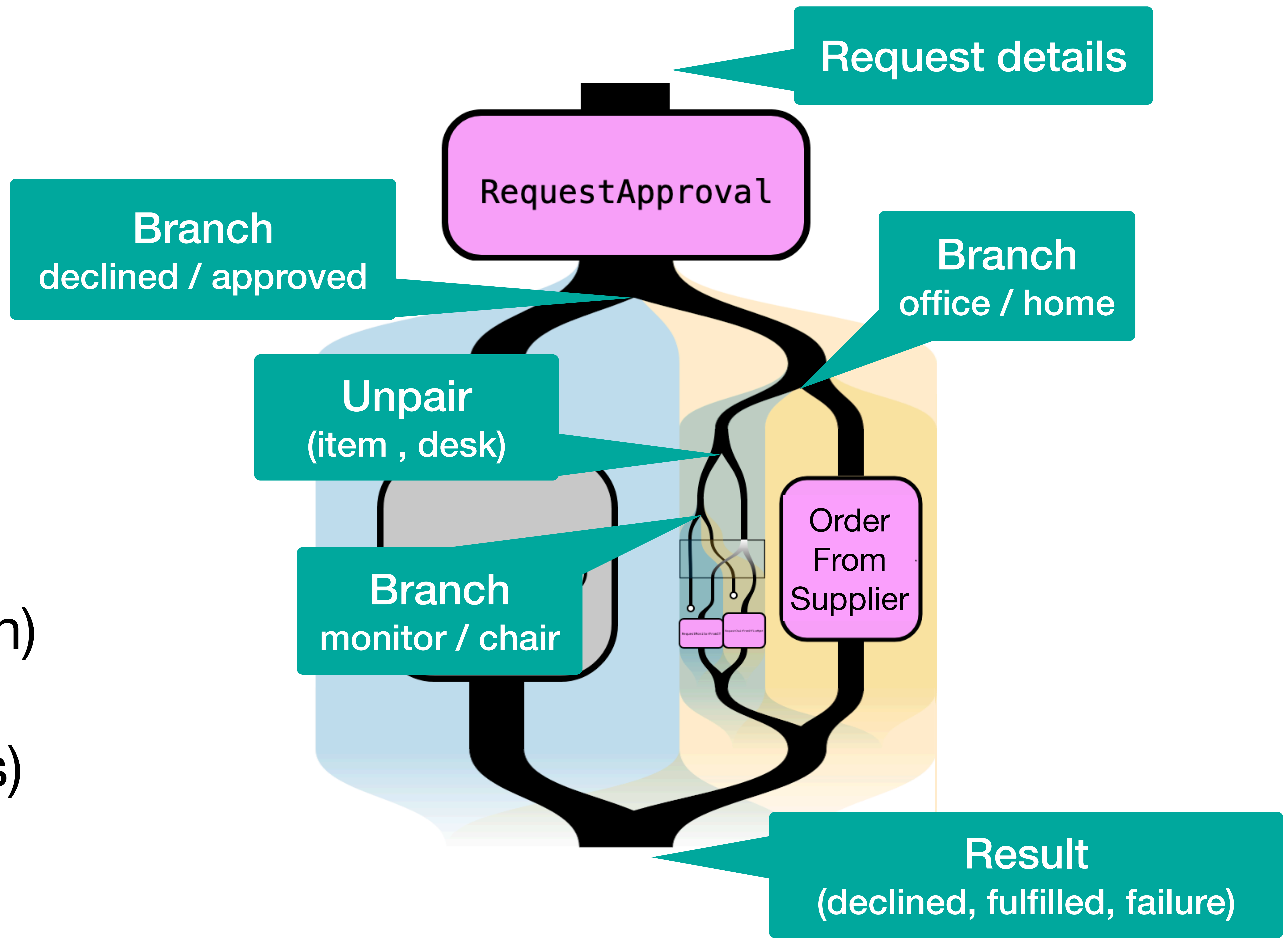
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

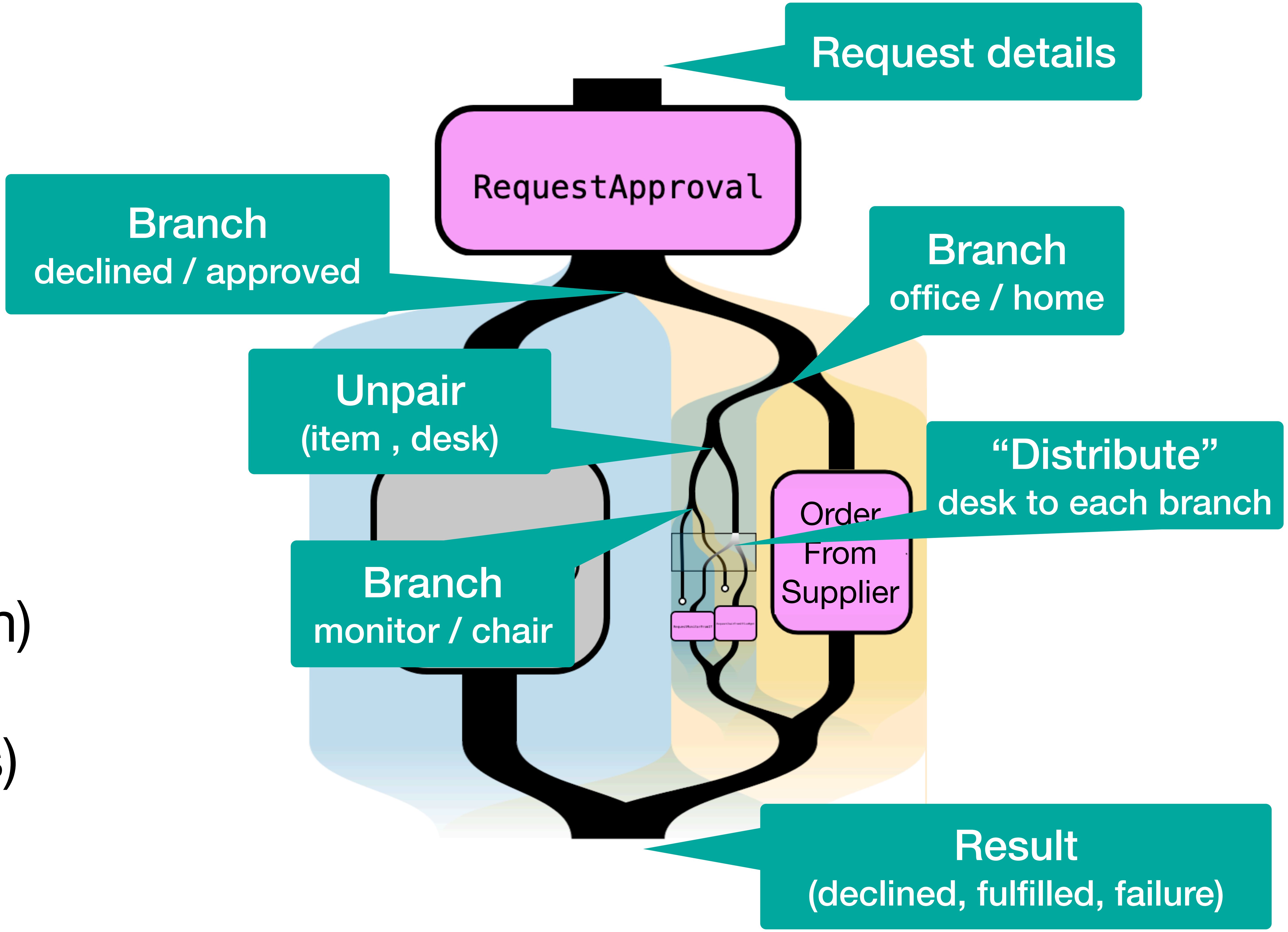
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

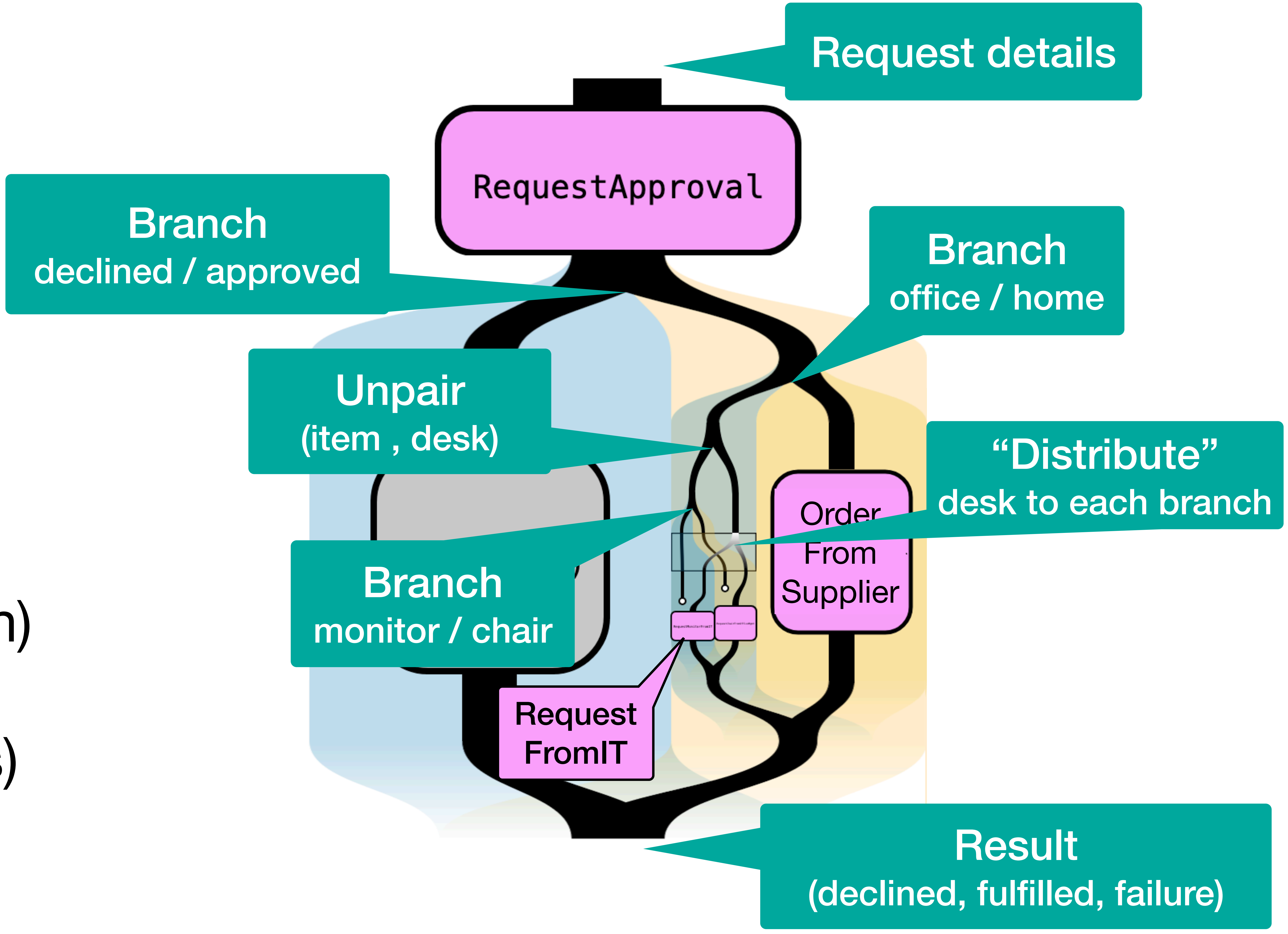
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

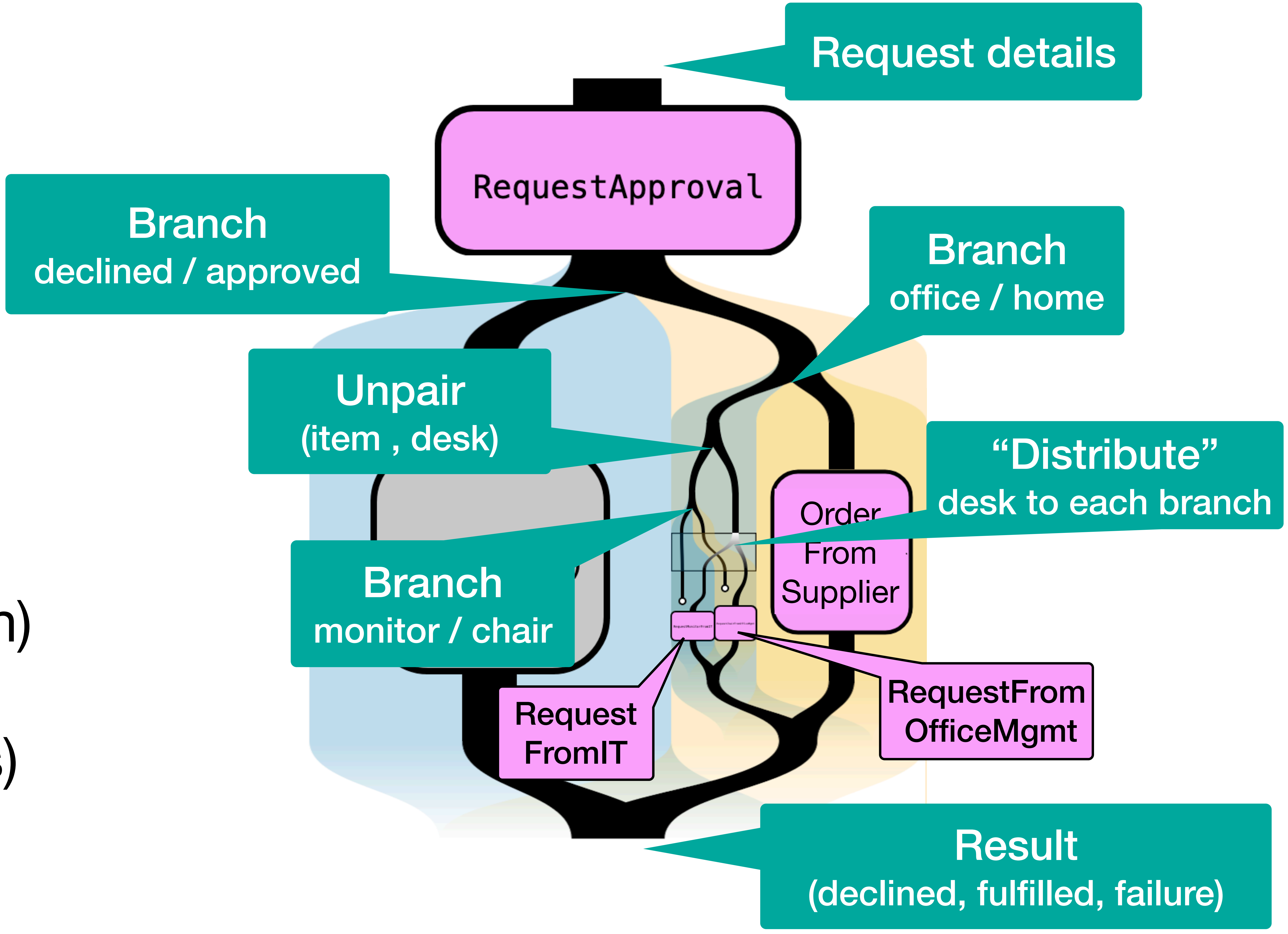
- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Example: Equipment Request

- Request work equipment
  - monitor, chair
- For use
  - In the office (+ desk location)
  - at home (+ delivery address)



(mostly) generated from code

Gathering Requirements

Gathering Requirements

- Expressive **control flow**

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**

Gathering Requirements

- Expressive **control flow**
    - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**

Gathering Requirements

- Expressive **control flow**
    - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state

Gathering Requirements

- Expressive **control flow**
    - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
    - **Serializable** state
- **Alternative interpretations**

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor

Gathering Requirements

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Others Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal Cadence Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal Cadence restate Do?

- Expressive **control flow**
    - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
    - **Serializable** state
- **Alternative interpretations**
    - Visualization, Simulation, …
- **Migration path**
    - to potential External DSL, Graphical Editor
    - *without rewriting* existing workflows

What Do Temporal Cadence restate GOLEM Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal / Cadence / restate / GOLEM / Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal, Cadence, restate, GOLEM, Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal Cadence restate GOLEM Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - **Serializable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

Just use the **host language**

What Do Temporal, Cadence, restate, GOLEM, Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**

Just use the **host language**

- **Durable** execution
  - ~~**Serializable** state~~ **Recoverable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

What Do Temporal Cadence restate GOLEM Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - ~~Serializable state~~ **Recoverable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

Just use the **host language**

& **event sourcing**

- log result of each activity
- **recovery:** restart,
  but use recorded activity results

What Do Temporal Cadence restate GOLEM Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - ~~**Serializable** state~~ 💡 **Recoverable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

Just use the **host language**

& **event sourcing**

- log result of each activity
- **recovery:** restart,
  but use recorded activity results

What Do Temporal, Cadence, restate, GOLEM, Workflows4s Do?

- Expressive **control flow**
  - Branching, Loops, Concurrency
- User-defined **functions**
- User-defined **data types**
- **Durable** execution
  - ~~Serializable state~~ 💡 **Recoverable** state
- **Alternative interpretations**
  - Visualization, Simulation, …
- **Migration path**
  - to potential External DSL, Graphical Editor
  - *without rewriting* existing workflows

Just use the **host language**

& **event sourcing**

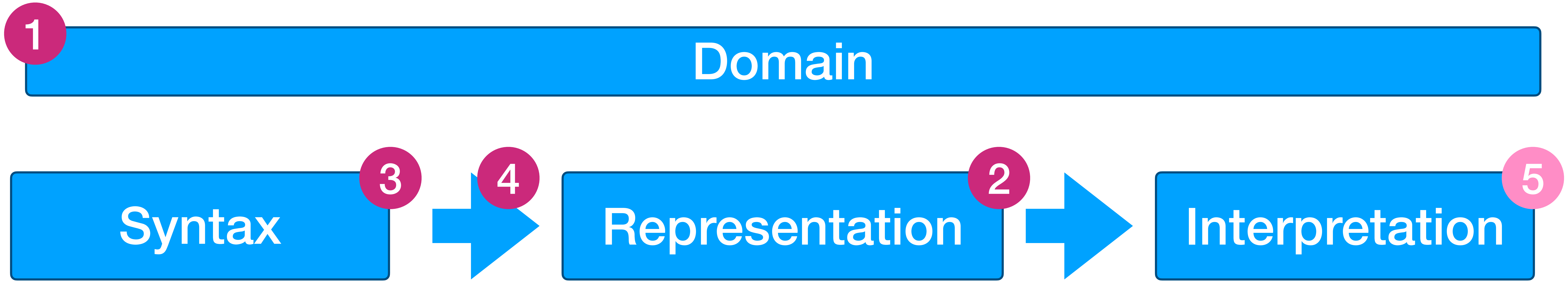
- log result of each activity
- **recovery:** restart,
  but use recorded activity results

Off-limits

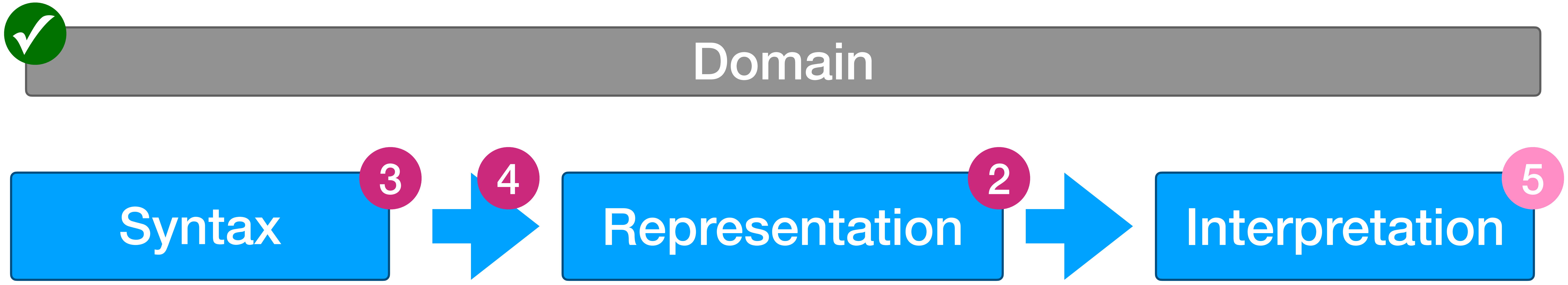
Not in control of representation.

There's only a single interpretation
of the host language: running it.

Agenda



Agenda



Roles

Roles

Language
Developer

design & implement
the Workflow DSL

Roles




This is you!

Language
Developer

design & implement
the Workflow DSL

Roles

```
import libretto.lambda.Lambdas
object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)
  opaque type Expr[A] = lambdas.Expr[A]
  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
```

This is you!

Language Developer

design & implement the Workflow DSL

Roles

```
import libretto.lambda.Lambdas
object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)
  opaque type Expr[A] = lambdas.Expr[A]
  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
```

This is you!

Workflow Developer

create workflows *using* the DSL

Language Developer

design & implement the Workflow DSL

Roles

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

Workflow Developer

create workflows *using* the DSL

```
import libretto.lambda.Lambdas
object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)
  opaque type Expr[A] = lambdas.Expr[A]
  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
```

This is you!

Language Developer

design & implement the Workflow DSL

Roles

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

```
import libretto.lambda.Lambdas
object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)
  opaque type Expr[A] = lambdas.Expr[A]
  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
```

Workflow User

Business person

Workflow Developer

create workflows *using* the DSL

This is you!

Language Developer

design & implement the Workflow DSL

Roles

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

```
import libretto.lambda.Lambdas
object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)
  opaque type Expr[A] = lambdas.Expr[A]
  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
```

This is you!

Workflow User	Workflow Developer	Language Developer	Library Code
Business person	create workflows *using* the DSL	design & implement the Workflow DSL	reusable bits from libretto-lambda

Roles

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

```
import libretto.lambda.Lambdas
object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)
  opaque type Expr[A] = lambdas.Expr[A]
  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
```

```
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val a : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(a)
  eliminate(a, from = b)
```

Workflow User

Business person

Workflow Developer

create workflows *using* the DSL

This is you!

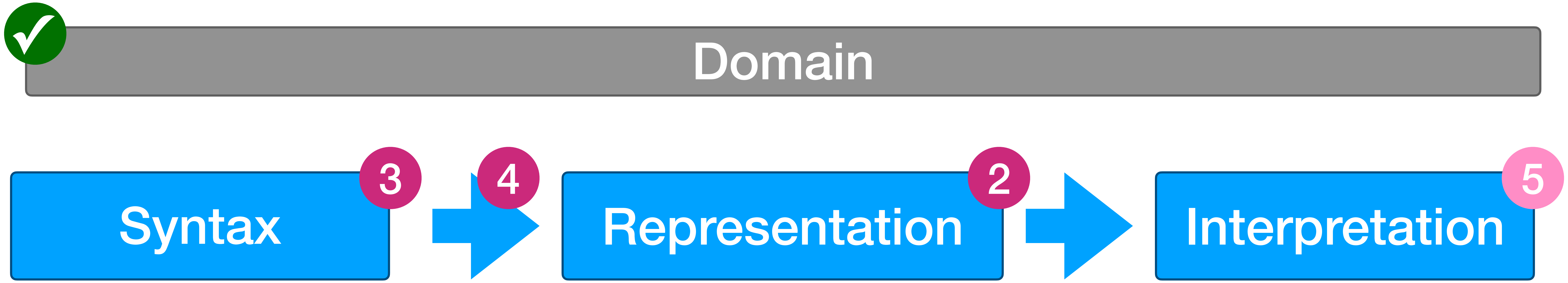
Language Developer

design & implement the Workflow DSL

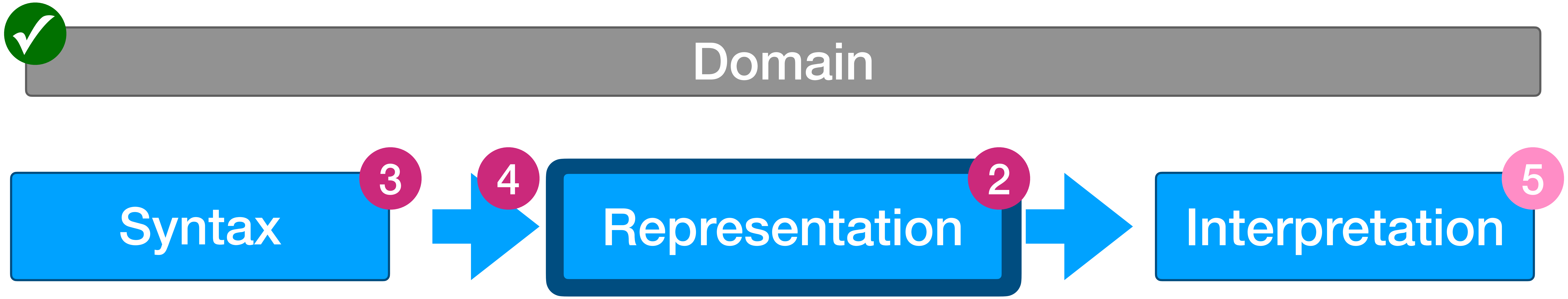
Library Code

reusable bits from libretto-lambda

Agenda



Agenda



Representing (eDSL) Programs

Choosing a Suitable Data Structure *(“AST”)*

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                        extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

straightforward

```
enum Expr[A]:
  case IntConstant(i: Int)                   extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                        extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                       extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads

Representing (eDSL) Programs

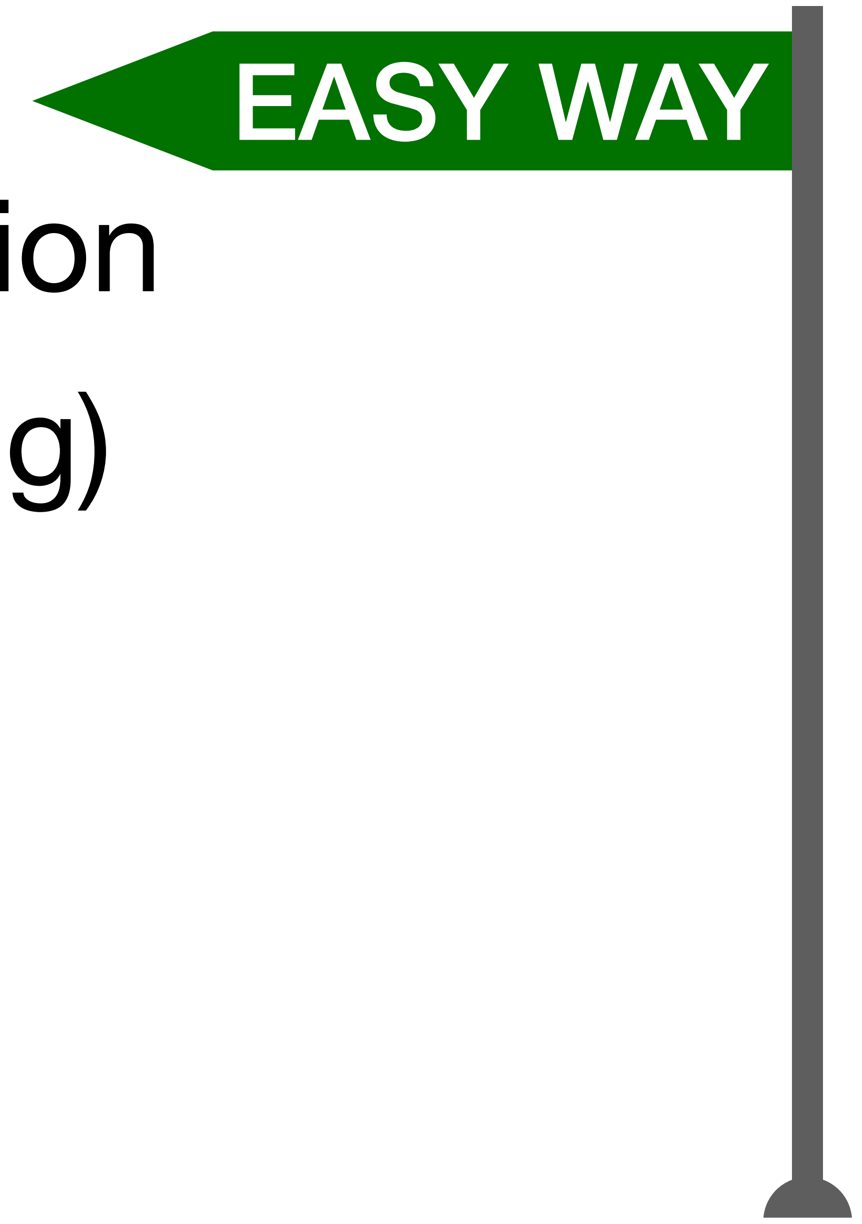
Choosing a Suitable Data Structure *("AST")*

```scala
enum Expr[A]:
  case IntConstant(i: Int)                      extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads



Just use a Scala function

(***"shallow"*** embedding)

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                        extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads

EASY WAY

Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

Representing (eDSL) Programs

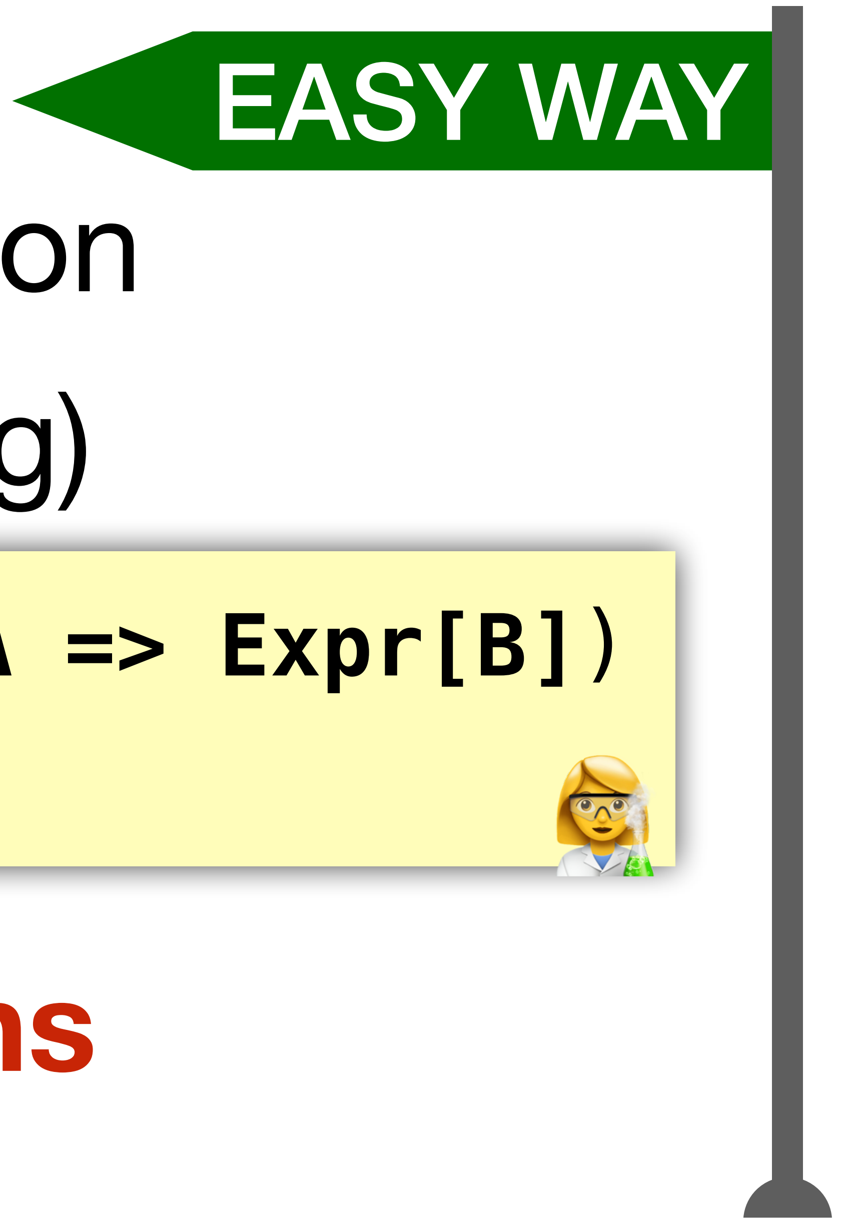
Choosing a Suitable Data Structure (*"AST"*)

```
enum Expr[A]:
  case IntConstant(i: Int)                       extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads



Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                    extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads

EASY WAY

Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

DEAD END

Representing (eDSL) Programs

Choosing a Suitable Data Structure (*“AST”*)

straightforward

until you need functions

```
enum Expr[A]:
  case IntConstant(i: Int)                      extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

Gain-of-functions crossroads

EASY WAY | RIGHT WAY

Just use a Scala function

(***“shallow”*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

DEAD END

deviantart.com/fernandesvincent

Functions as Data

(***“deep”*** embedding)

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                    extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads

Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

deviantart.com/fernandesvincent

Functions as Data

(***"deep"*** embedding)

- no Scala functions inside

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

```
enum Expr[A]:
  case IntConstant(i: Int)                      extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

straightforward

until you need functions

Gain-of-functions crossroads

EASY WAY | RIGHT WAY

Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

DEAD END

deviantart.com/fernandesvincent

Functions as Data

(***"deep"*** embedding)

- no Scala functions inside
- fully introspectable

Representing (eDSL) Programs

Choosing a Suitable Data Structure (*"AST"*)

straightforward

until you need functions

```
enum Expr[A]:
  case IntConstant(i: Int)                      extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

Gain-of-functions crossroads

EASY WAY | RIGHT WAY

Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

DEAD END

deviantart.com/fernandesvincent

Functions as Data

(***"deep"*** embedding)

- no Scala functions inside
- fully introspectable
- *we* are **in control**

Representing (eDSL) Programs

Choosing a Suitable Data Structure *("AST")*

straightforward

until you need functions

```
enum Expr[A]:
  case IntConstant(i: Int)                  extends Expr[Int]
  case Plus(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
  ...
```

Gain-of-functions crossroads

EASY WAY | RIGHT WAY

Just use a Scala function

(***"shallow"*** embedding)

```
case FlatMap[A, B](a: Expr[A], f: A => Expr[B])
  extends Expr[B]
```

no alternative interpretations

DEAD END

deviantart.com/fernandesvincent

Functions as Data

(***"deep"*** embedding)

- no Scala functions inside
- fully introspectable
- *we* are **in control**

Representing Functions

Representing Functions

Representing Functions

Expression-centric

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])     extends Expr[Int]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])      extends Expr[Int]
  case Var[A](name: String)                  extends Expr[A]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])      extends Expr[Int]
  case Var[A](name: String)                  extends Expr[A]
  case Lam[A,B](a: Var[A], b: Expr[B])       extends Expr[A ⇒ B]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])      extends Expr[Int]
  case Var[A](name: String)                  extends Expr[A]
  case Lam[A,B](a: Var[A], b: Expr[B])       extends Expr[A ⇒ B]
  case App[A,B](f: Expr[A ⇒ B], a: Expr[A]) extends Expr[B]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])       extends Expr[Int]
  case Var[A](name: String)                   extends Expr[A]
  case Lam[A,B](a: Var[A], b: Expr[B])        extends Expr[A ⇒ B]
  case App[A,B](f: Expr[A ⇒ B], a: Expr[A]) extends Expr[B]
```

```
x ⇒ f(x) + g(x)
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])     extends Expr[Int]
  case Var[A](name: String)                 extends Expr[A]
  case Lam[A,B](a: Var[A], b: Expr[B])      extends Expr[A ⇒ B]
  case App[A,B](f: Expr[A ⇒ B], a: Expr[A]) extends Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  case Plus(l: Expr[Int], r: Expr[Int])      extends Expr[Int]
  case Var[A](name: String)                  extends Expr[A]
  case Lam[A,B](a: Var[A], b: Expr[B])       extends Expr[A ⇒ B]
  case App[A,B](f: Expr[A ⇒ B], a: Expr[A]) extends Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Representing Functions

Expression-centric

Function-centric
(Point-free)

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
enum Flow[A,B]:
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

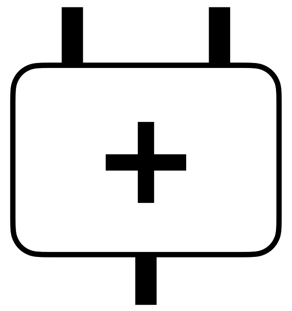
```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Function-centric
(Point-free)

```
enum Flow[A,B]:
  case Plus()
    extends Flow[(Int, Int), Int]
```



Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

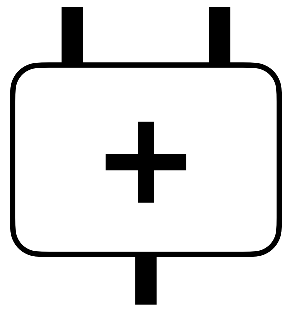
```
val f, g: Expr[Int ⇒ Int] = ???
```

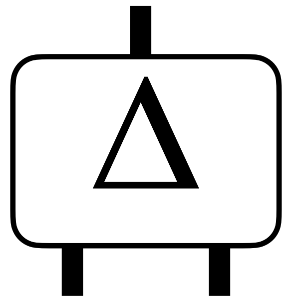
```
x ⇒ f(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Function-centric
(Point-free)

```
enum Flow[A,B]:
  case Plus()
    extends Flow[(Int, Int), Int]
  case Dup[A]()
    extends Flow[A, (A, A)]
```





Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

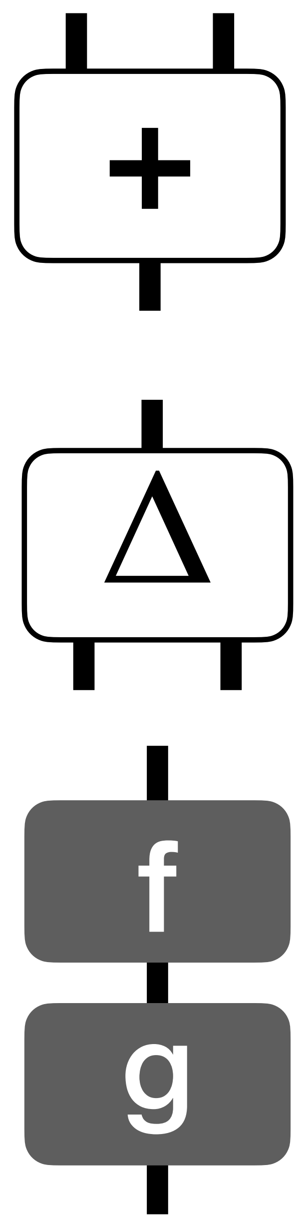
```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Function-centric
(Point-free)

```
enum Flow[A,B]:
  case Plus()
    extends Flow[(Int, Int), Int]
  case Dup[A]()
    extends Flow[A, (A, A)]
  case AndThen[A,B,C](
    f: Flow[A,B],
    g: Flow[B,C]
  ) extends Flow[A,C]
```



Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

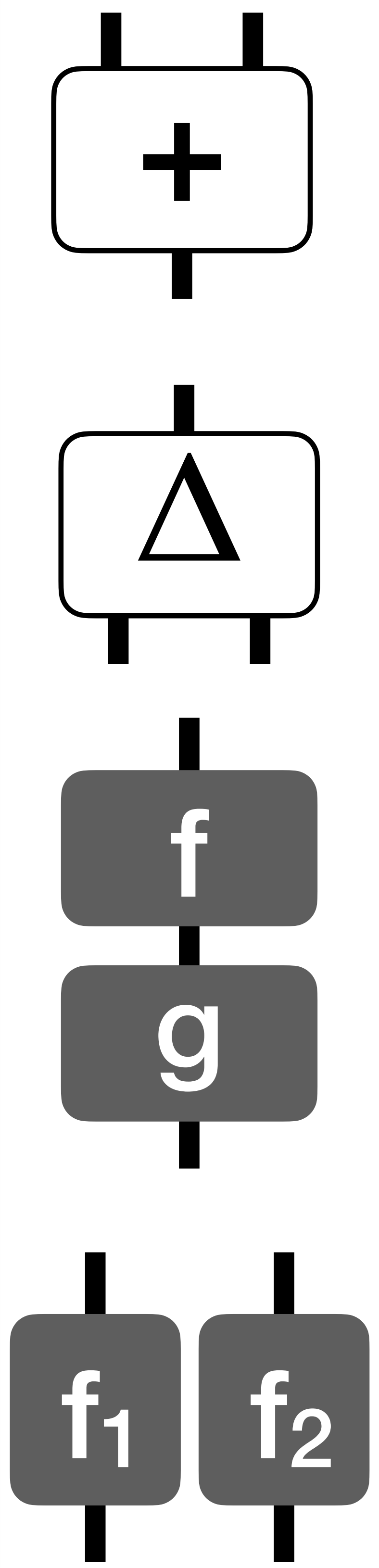
```
x ⇒ f(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Function-centric

(Point-free)

```
enum Flow[A,B]:
  case Plus()
    extends Flow[(Int, Int), Int]
  case Dup[A]()
    extends Flow[A, (A, A)]
  case AndThen[A,B,C](
    f: Flow[A,B],
    g: Flow[B,C]
  ) extends Flow[A,C]
  case Par[A1, A2, B1, B2](
    f1: Flow[A1, B1],
    f2: Flow[A2, B2]
  ) extends Flow[(A1, A2), (B1, B2)]
```



Representing Functions

Expression-centric

Function-centric
(Point-free)

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
enum Flow[A,B]:
  Plus(): Flow[(Int,Int), Int]
  Dup() : Flow[A, (A,A)]
  AndThen(Flow[A,B], Flow[B,C]): Flow[A,C]
  Par(Flow[A₁,B₁], Flow[A₂,B₂])
   : Flow[(A₁,A₂), (B₁,B₂)]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

Function-centric

(Point-free)

```
enum Flow[A,B]:
  Plus(): Flow[(Int,Int), Int]
  Dup() : Flow[A, (A,A)]
  AndThen(Flow[A,B], Flow[B,C]): Flow[A,C]
  Par(Flow[A1,B1], Flow[A2,B2])
   : Flow[(A1,A2), (B1,B2)]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

Function-centric

(Point-free)

```
enum Flow[A,B]:
  Plus(): Flow[(Int,Int), Int]
  Dup() : Flow[A, (A,A)]
  AndThen(Flow[A,B], Flow[B,C]): Flow[A,C]
  Par(Flow[A1,B1], Flow[A2,B2])
   : Flow[(A1,A2), (B1,B2)]
```

```
val f, g: Flow[Int, Int] = ???
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Representing Functions

Expression-centric

Function-centric
(Point-free)

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
enum Flow[A,B]:
  Plus(): Flow[(Int,Int), Int]
  Dup() : Flow[A, (A,A)]
  AndThen(Flow[A,B], Flow[B,C]): Flow[A,C]
  Par(Flow[A1,B1], Flow[A2,B2])
   : Flow[(A1,A2), (B1,B2)]
```

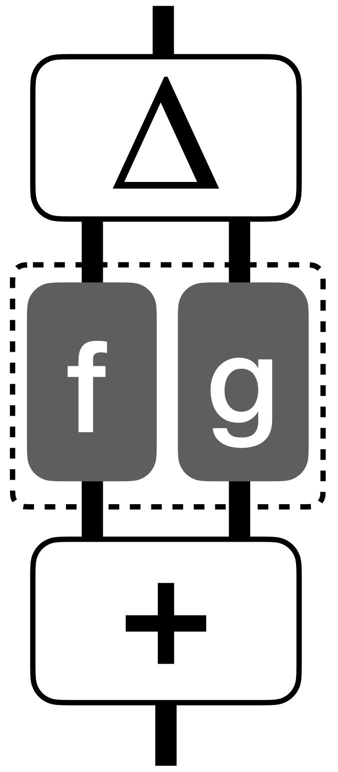
```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```



Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

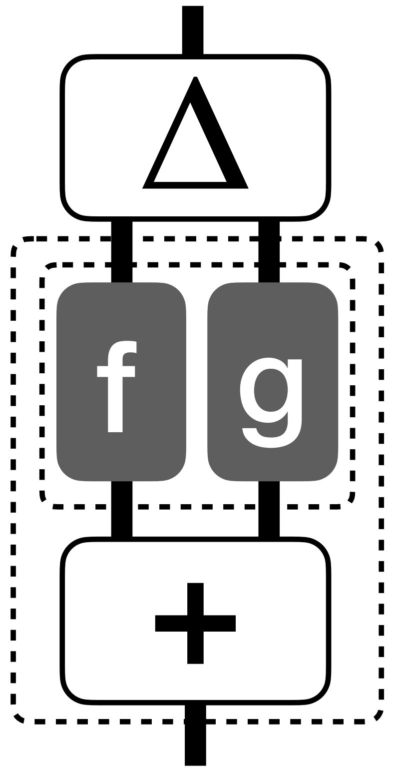
Function-centric

(Point-free)

```
enum Flow[A,B]:
  Plus(): Flow[(Int,Int), Int]
  Dup() : Flow[A, (A,A)]
  AndThen(Flow[A,B], Flow[B,C]): Flow[A,C]
  Par(Flow[A1,B1], Flow[A2,B2])
   : Flow[(A1,A2), (B1,B2)]
```

```
val f, g: Flow[Int, Int] = ???
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```



Representing Functions

Expression-centric

```
enum Expr[A]:
  Plus(Expr[Int], Expr[Int]): Expr[Int]
  Var(String)               : Expr[A]
  Lam(Var[A], Expr[B])      : Expr[A ⇒ B]
  App(Expr[A ⇒ B], Expr[A]) : Expr[B]
```

Function-centric
(Point-free)

```
enum Flow[A,B]:
  Plus(): Flow[(Int,Int), Int]
  Dup() : Flow[A, (A,A)]
  AndThen(Flow[A,B], Flow[B,C]): Flow[A,C]
  Par(Flow[A1,B1], Flow[A2,B2])
   : Flow[(A1,A2), (B1,B2)]
```

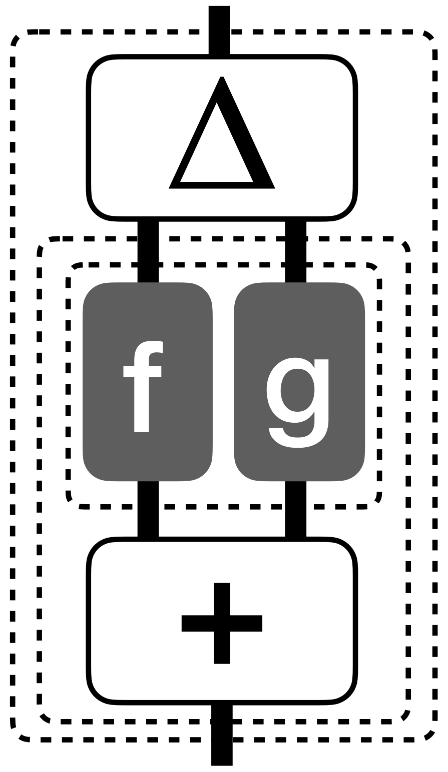
```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

🟢 close to syntax

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

🔴 needs translation

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives

Expression centric vs. Function centric

```
x ⇒ f(x) + g(x)
```

```
val f, g: Expr[Int ⇒ Int] = ???
```

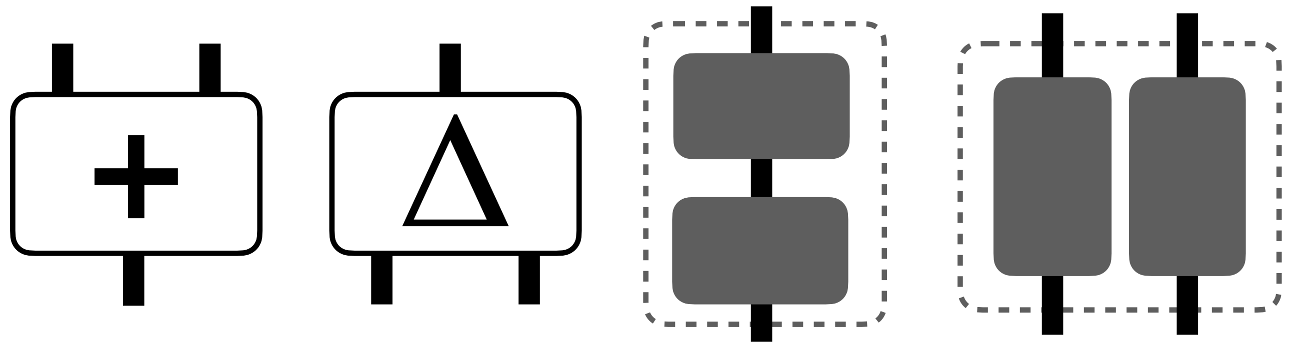
- 🟢 close to syntax
- 🟢 few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
val f, g: Flow[Int, Int] = ???
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- 🔴 needs translation
- 🔴 many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

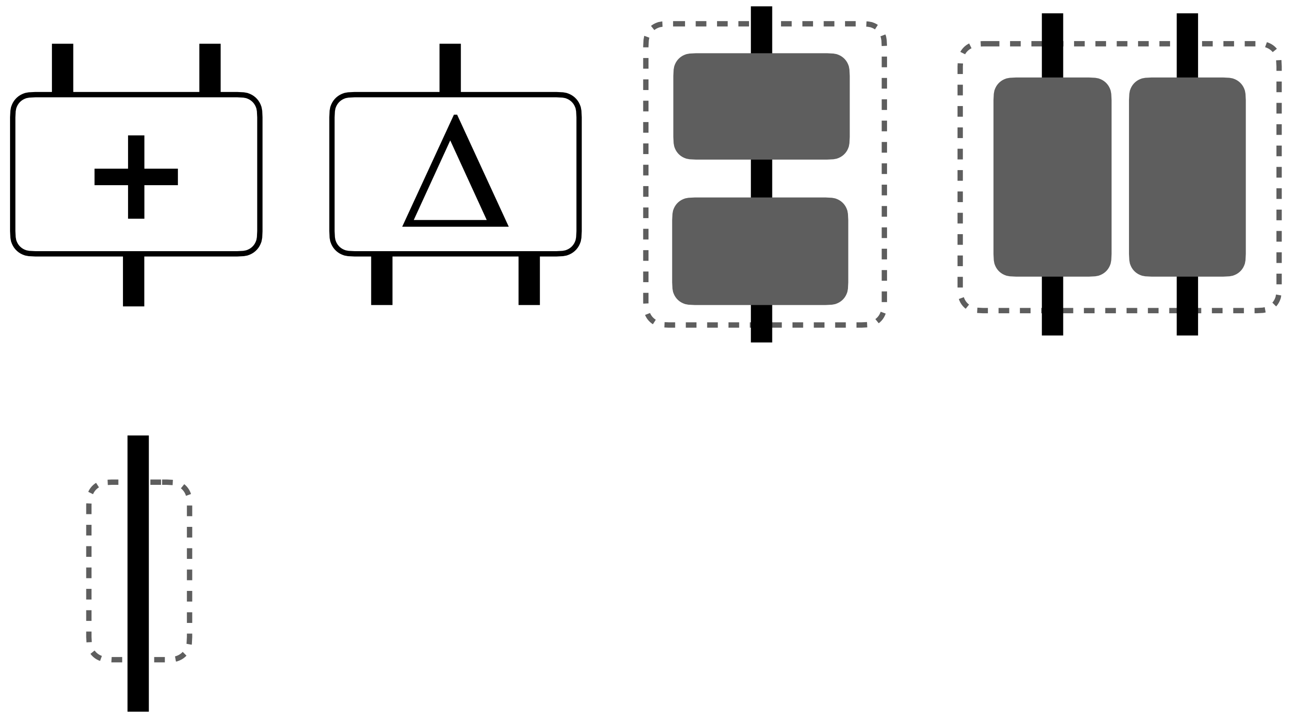
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

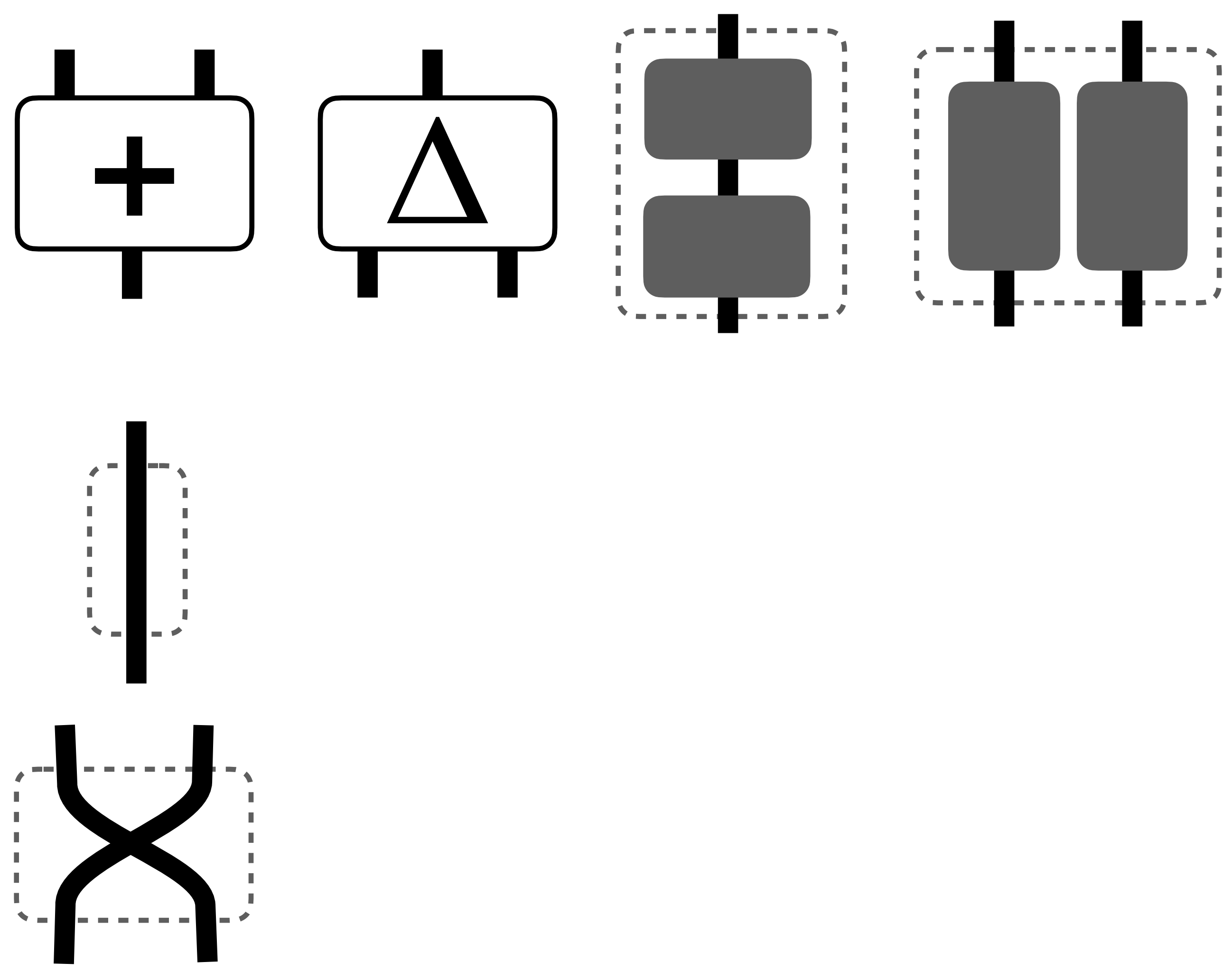
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

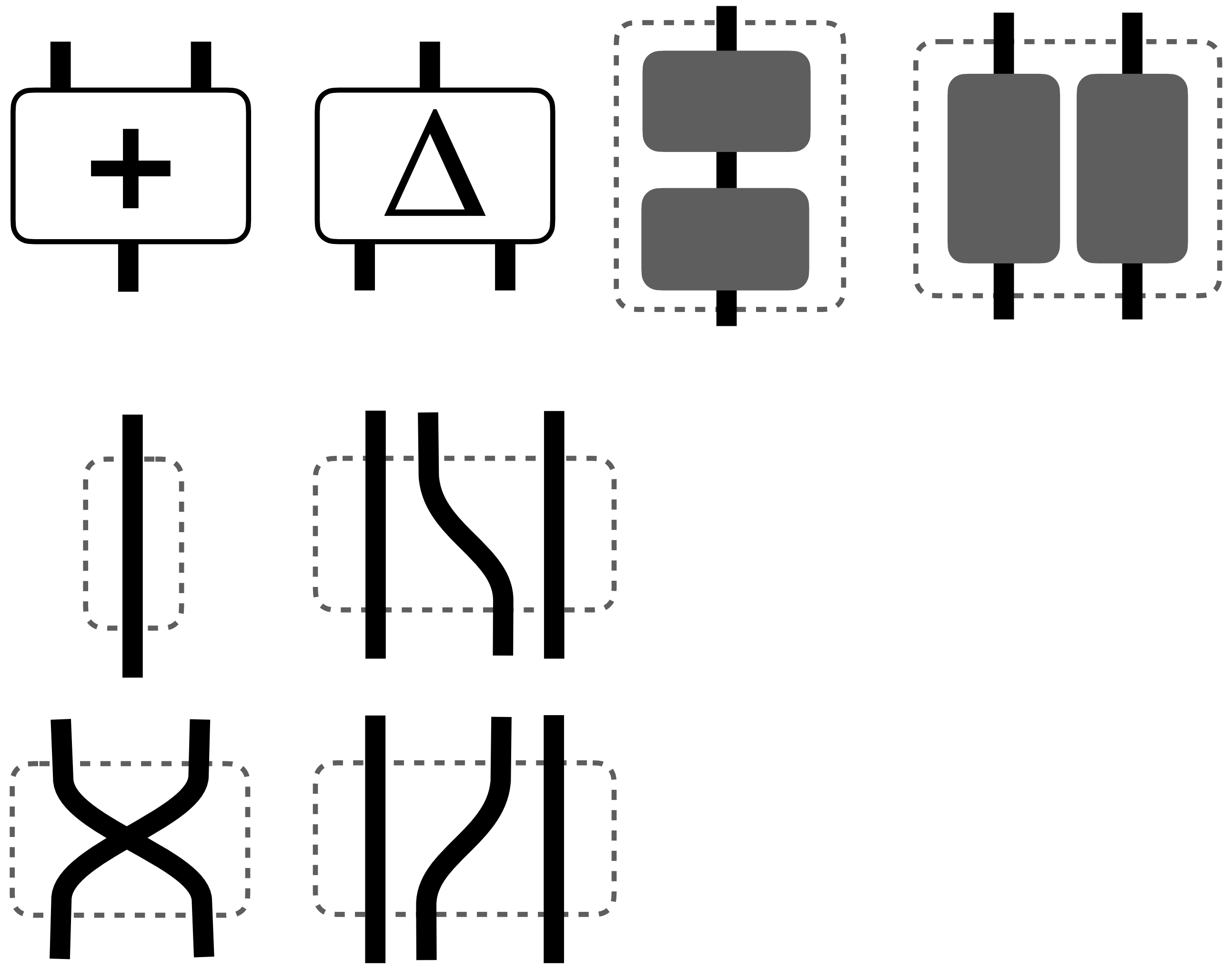
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

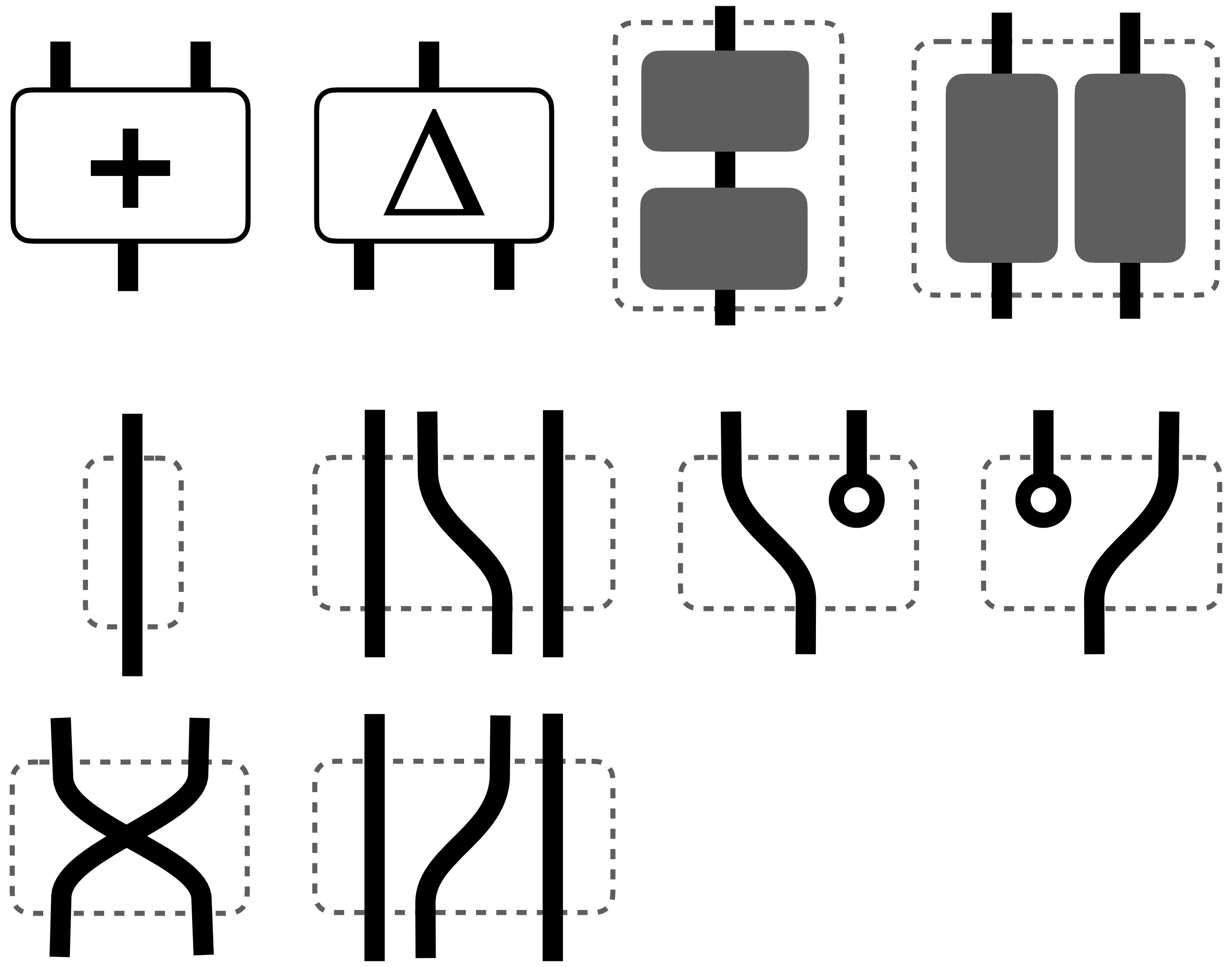
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

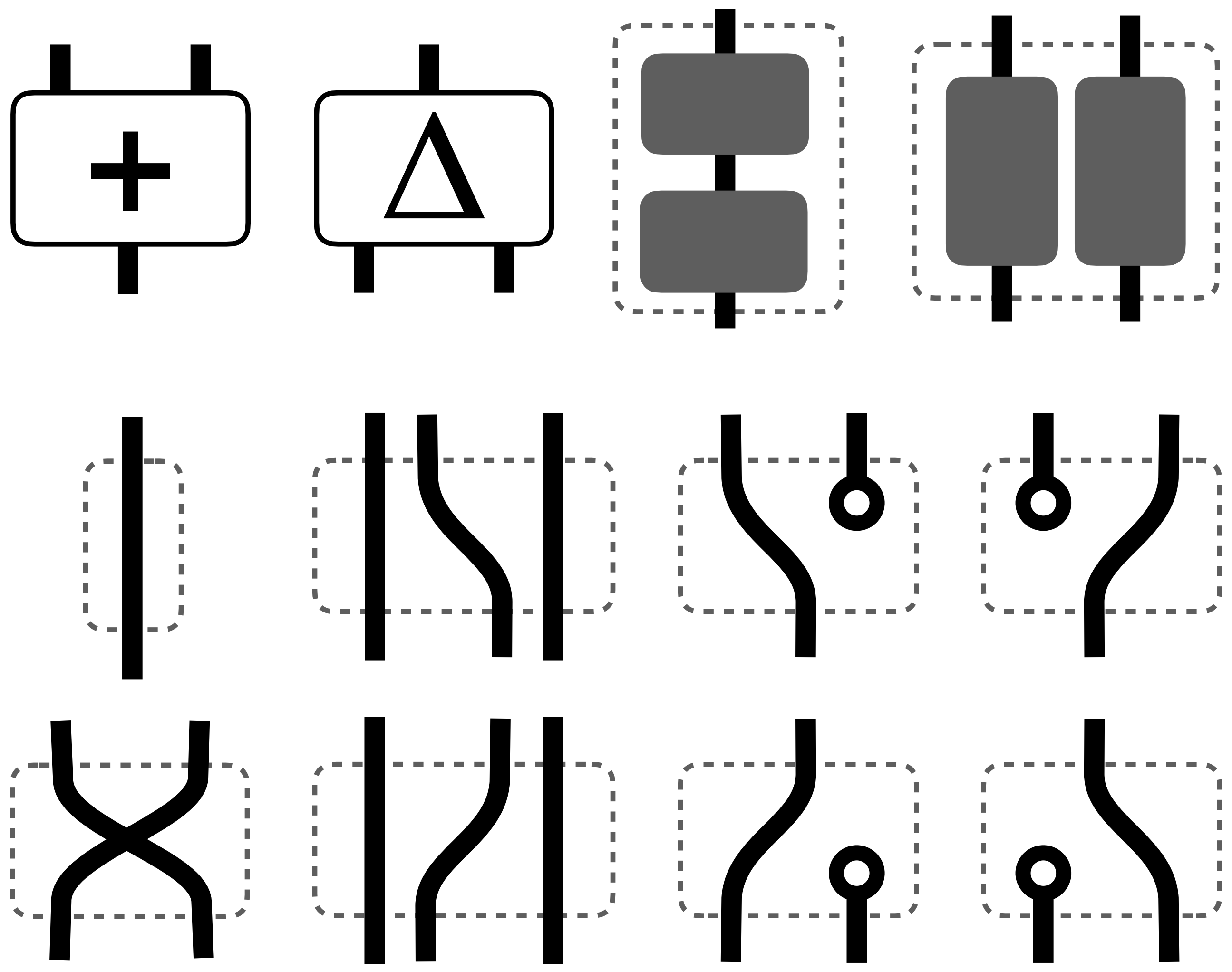
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

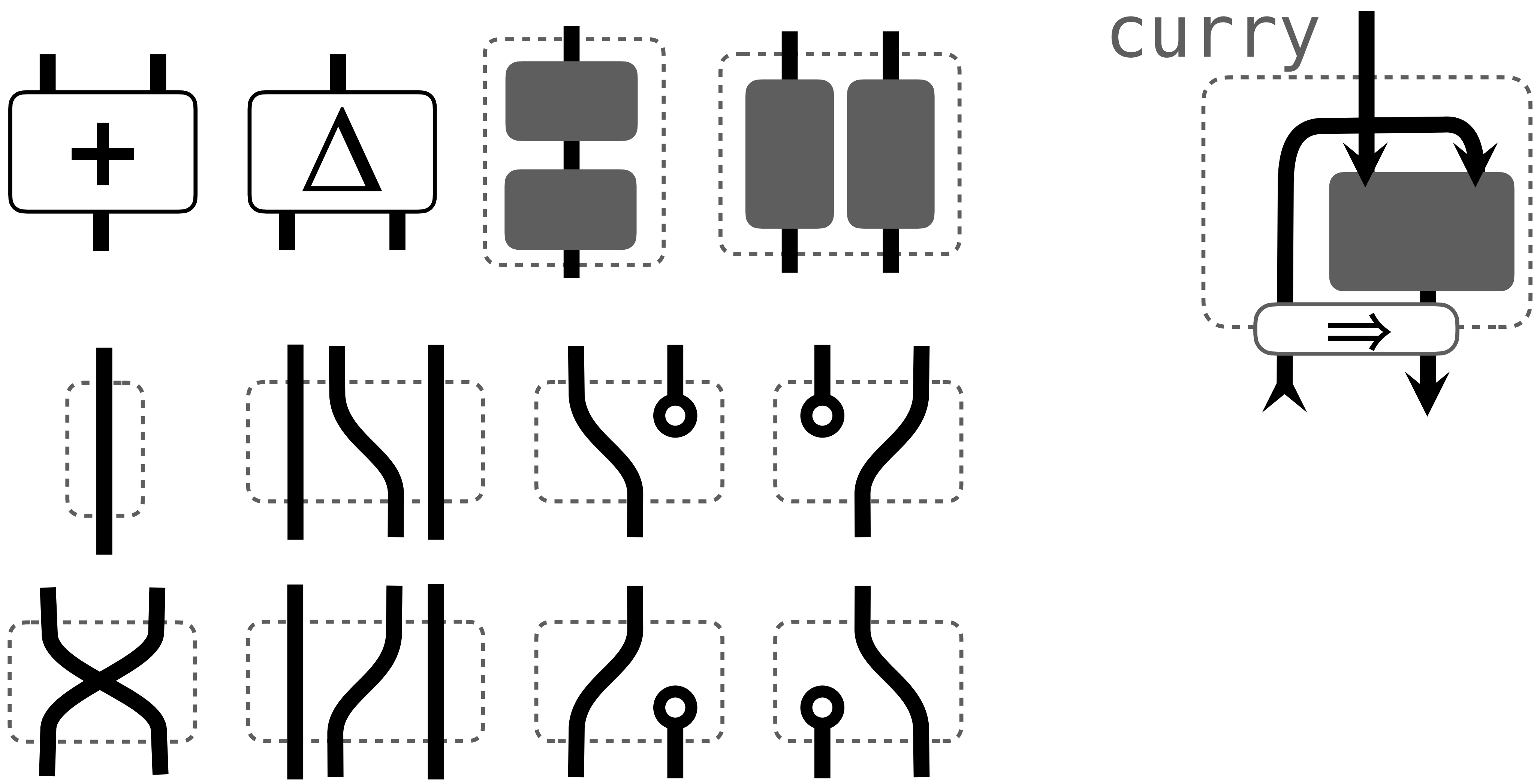
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

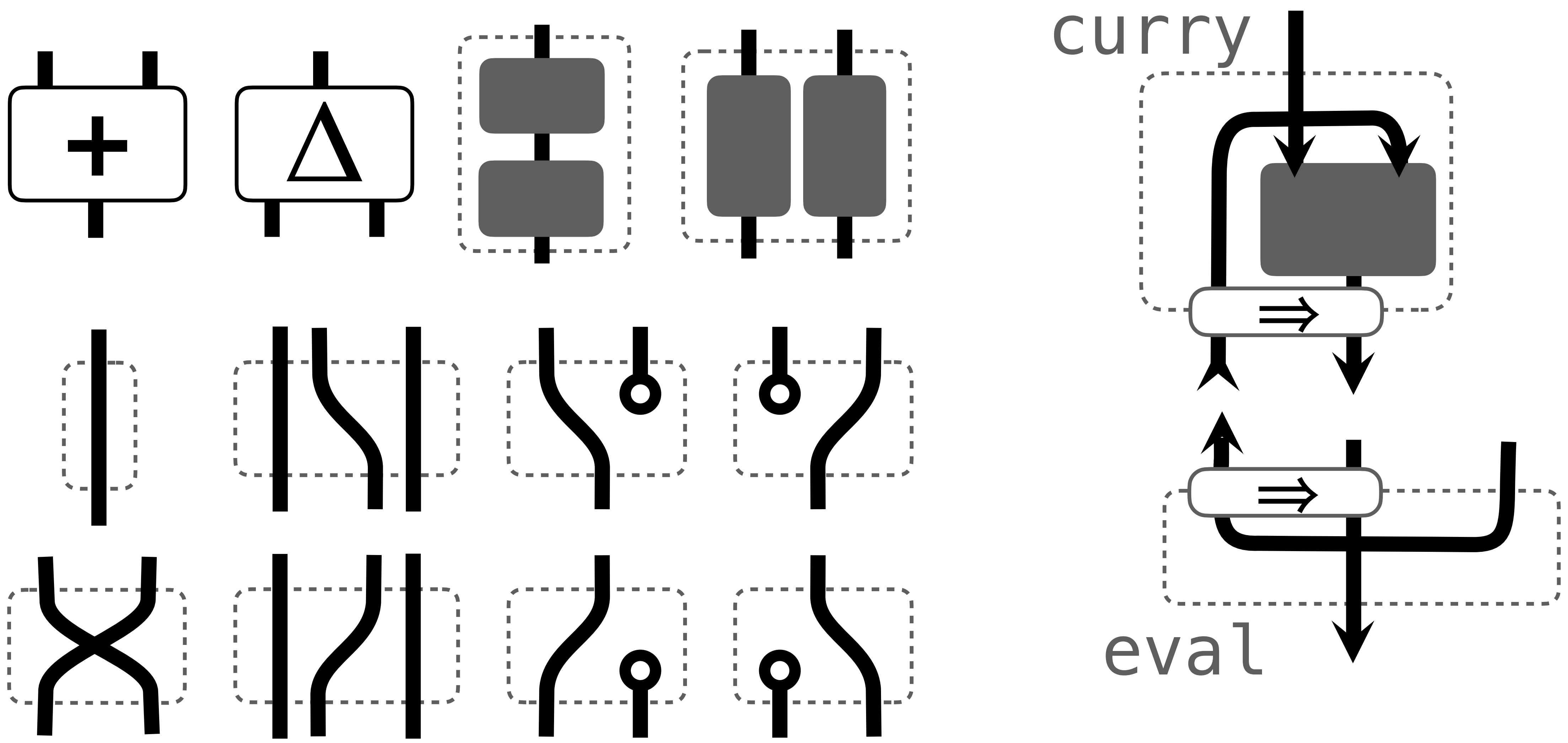
```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives



Expression centric vs. Function centric

```scala
val f, g: Expr[Int ⇒ Int] = ???
```

```scala
x ⇒ f(x) + g(x)
```

```scala
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise

```scala
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```scala
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose
- ~ category theory

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose
- ~ category theory

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose
- ~ category theory
- values as functions

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values
  - Is a *world-capturing* closure still a *value?*

- needs translation
- many primitives
- verbose
- ~ category theory
- values as functions

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- 🟢 close to syntax
- 🟢 few primitives
- 🟢 concise
- ⚫ ~ λ-calculus
- ⚫ functions as values
  - 🔴 Is a *world-capturing* closure still a *value?*
- 🔴 non-locality (referencing arbitrarily distant variables)

- 🔴 needs translation
- 🔴 many primitives
- 🔴 verbose
- ⚫ ~ category theory
- ⚫ values as functions

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Expression centric:

- 🟢 close to syntax
- 🟢 few primitives
- 🟢 concise
- ⚫ ~ λ-calculus
- ⚫ functions as values
  - 🔴 Is a *world-capturing* closure still a *value?*
- 🔴 non-locality (referencing arbitrarily distant variables)

Function centric:

- 🔴 needs translation
- 🔴 many primitives
- 🔴 verbose
- ⚫ ~ category theory
- ⚫ values as functions
- 🟢 locality (everything discoverable by *"wire chasing"*)

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Expression centric:

- 🟢 close to syntax
- 🟢 few primitives
- 🟢 concise
- ⚫ ~ λ-calculus
- ⚫ functions as values
  - 🔴 Is a *world-capturing* closure still a *value?*
- 🔴 non-locality (referencing arbitrarily distant variables)
- 🔴 all-or-nothing
  - HOFs/closures, non-linearity, Church encodings, …
  - can't meaningfully take away any of `Var`, `Lam`, `App`

Function centric:

- 🔴 needs translation
- 🔴 many primitives
- 🔴 verbose
- ⚫ ~ category theory
- ⚫ values as functions
- 🟢 locality (everything discoverable by *"wire chasing"*)

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Expression centric:

- 🟢 close to syntax
- 🟢 few primitives
- 🟢 concise
- ⚫ ~ λ-calculus
- ⚫ functions as values
  - 🔴 Is a *world-capturing* closure still a *value?*
- 🔴 non-locality (referencing arbitrarily distant variables)
- 🔴 all-or-nothing
  - HOFs/closures, non-linearity, Church encodings, …
  - can't meaningfully take away any of `Var`, `Lam`, `App`

Function centric:

- 🔴 needs translation
- 🔴 many primitives
- 🔴 verbose
- ⚫ ~ category theory
- ⚫ values as functions
- 🟢 locality (everything discoverable by *"wire chasing"*)
- 🟢 graded expressive power
  - pairs before HOFs; don't have to have HOFs
  - linearity by *taking away* non-linear ops

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values
- non-locality
- all-or-nothing

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose
- ~ category theory
- values as functions
- locality
- graded expressive power

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values
- non-locality
- all-or-nothing
- pervasive **illegal state**
  - undefined variables
  - shadowing
  - program transformations *inevitably* deal with illegal fragments

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose
- ~ category theory
- values as functions
- locality
- graded expressive power

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

Expression centric

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values
- non-locality
- all-or-nothing
- pervasive **illegal state**
  - undefined variables
  - shadowing
  - program transformations *inevitably* deal with illegal fragments

Function centric

- needs translation
- many primitives
- verbose
- ~ category theory
- values as functions
- locality
- graded expressive power
- illegal state largely unrepresentable
  - no variables ⇒ no *undefined* variables or shadowing
  - no illegal fragments

Expression centric vs. Function centric

```
val f, g: Expr[Int ⇒ Int] = ???
```

```
x ⇒ f(x) + g(x)
```

```
val f, g: Flow[Int, Int] = ???
```

```
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- 🟢 close to syntax
- 🟢 few primitives
- 🟢 concise
- ⚫ ~ λ-calculus
- ⚫ functions as values
- 🔴 non-locality
- 🔴 all-or-nothing
- 🔴 pervasive **illegal state**
  - undefined variables
  - shadowing
  - program transformations *inevitably* deal with illegal fragments

- 🔴 needs translation
- 🔴 many primitives
- 🔴 verbose
- ⚫ ~ category theory
- ⚫ values as functions
- 🟢 locality
- 🟢 graded expressive power
- 🟢 illegal state largely unrepresentable
  - no variables ⇒ no *undefined* variables or shadowing
  - no illegal fragments
- 🟢 graphical notation

Expression centric vs. Function centric

```scala
val f, g: Expr[Int ⇒ Int] = ???
```

```scala
x ⇒ f(x) + g(x)
```

```scala
val f, g: Flow[Int, Int] = ???
```

- close to syntax
- few primitives
- concise
- ~ λ-calculus
- functions as values
- non-locality
- all-or-nothing

```scala
Lam(
  Var("x"),
  Plus(
    App(f, Var("x")),
    App(g, Var("x"))
  )
): Expr[Int ⇒ Int]
```

```scala
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    Plus()
  )
): Flow[Int, Int]
```

- needs translation
- many primitives
- verbose
- ~ category theory
- values as functions
- locality
- graded expressive power

- pervasive **illegal state**
  - undefined variables
  - shadowing
  - program transformations *inevitably* deal with illegal fragments

- illegal state largely unrepresentable
  - no variables ⇒ no *undefined* variables or shadowing
  - no illegal fragments
- graphical notation

THIS WAY

Values as Functions

Values as Functions

```
case Const[A](value: A) extends Flow[Unit, A]
```

Values as Functions

```scala
case Const[A](value: A) extends Flow[Unit, A]
```

```scala
Const(new Thread()) : Flow[Unit, Thread]
```

Values as Functions

```
case Const[A](value: A) extends Flow[Unit, A]
```

```
Const(new Thread()) : Flow[Unit, Thread]
```

Values as Functions

```
case Const[A](value: A) extends Flow[Unit, A]
```

```
Const(new Thread()) : Flow[Unit, Thread]
```

A live thread inside AST?

Values as Functions

```scala
case Const[A](value: A) extends Flow[Unit, A]
```

```scala
Const(new Thread()) : Flow[Unit, Thread]
```

A live Thread inside AST?

Values as Functions

```scala
case Const[A](value: A) extends Flow[Unit, A]
```

```scala
Const(new Thread()) : Flow[Unit, Thread]
```

A live thread inside AST?

Values as Functions

```scala
case Const[A](value: A) extends Flow[Unit, A]
```

```scala
Const(new Thread()) : Flow[Unit, Thread]
```

A live thread inside AST?

```scala
case Const[A](value: Value[A]) extends Flow[Unit, A]
```

Values as Functions

case **Const**[A](~~value:~~ A) extends Flow[Unit, A]

```
Const(new Thread()) : Flow[Unit, Thread]
```

A live thread inside AST?

```
case Const[A](value: Value[A]) extends Flow[Unit, A]
```

A GADT that limits what is a domain-level value

Values as Functions

```
case Const[A](value: A) extends Flow[Unit, A]
```

```
Const(new Thread()) : Flow[Unit, Thread]
```

A live thread inside AST?

```
case Const[A](value: Value[A]) extends Flow[Unit, A]
```

A GADT that limits what is a domain-level value

```
enum Value[A]:
  case Integer(i: Int) extends Value[Int]
  case Pair[A, B](
    a: Value[A],
    b: Value[B]
  ) extends Value[(A, B)]
  ...
```

Values as Functions

```
case Const[A](value: A) extends Flow[Unit, A]
```

```
Const(new Thread()) : Flow[Unit, Thread]
```

A live thread inside AST?

```
case Const[A](value: Value[A]) extends Flow[Unit, A]
```

A GADT that limits what is a domain-level value

```
enum Value[A]:
  case Integer(i: Int) extends Value[Int]
  case Pair[A, B](
    a: Value[A],
    b: Value[B]
  ) extends Value[(A, B)]
  ...
```

```
Const(new Thread())
```

```
Found:    Thread
Required: Value[A]
```

Best Practice:

Don’t Reuse Scala Types as Domain Types

Best Practice:

Don’t Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Best Practice:
Don’t Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

Best Practice:
Don’t Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

- keep them **uninhabited** at Scala level

Best Practice:
Don’t Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

- keep them **uninhabited** at Scala level

```
sealed trait **[A, B] // no subclasses
```

Best Practice:
Don’t Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

- keep them **uninhabited** at Scala level

```
sealed trait **[A, B] // no subclasses
```

  - used only in phantom positions

Best Practice:
Don't Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

- keep them **uninhabited** at Scala level

```scala
sealed trait **[A, B] // no subclasses
```

- used only in phantom positions
- don't ask what it *is*, but what you can ***do*** with it

Best Practice:
Don't Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

- keep them **uninhabited** at Scala level

```
sealed trait **[A, B] // no subclasses
```

  - used only in phantom positions
  - don't ask what it *is*, but what you can ***do*** with it

```
enum Flow[A,B]:
  case Dup[A]() extends Flow[A, A ** A]
  case Par[A1, A2, B1, B2](
    f1: Flow[A1, B1],
    f2: Flow[A2, B2]
  ) extends Flow[A1 ** A2, B1 ** B2]
  case Const[A](value: Value[A]) extends Flow[One, A]
  ...
```

Best Practice:
Don't Reuse Scala Types as Domain Types

Introduce new domain types for common concepts

Scala	Domain (Workflows)
`(A, B)`	`A ** B`
`Either[A, B]`	`A ++ B`
`Unit`	`One`

- keep them **uninhabited** at Scala level

```
sealed trait **[A, B] // no subclasses
```

  - used only in phantom positions
  - don't ask what it *is*, but what you can ***do*** with it

```
enum Flow[A,B]:
  case Dup[A]() extends Flow[A, A ** A]
  case Par[A1, A2, B1, B2](
    f1: Flow[A1, B1],
    f2: Flow[A2, B2]
  ) extends Flow[A1 ** A2, B1 ** B2]
  case Const[A](value: Value[A]) extends Flow[One, A]
  ...
```

```
enum Value[A]:
  case Pair[A, B](
    a: Value[A],
    b: Value[B]
  ) extends Value[A ** B]
  ...
```

Domain-level Enums

(a.k.a. sum types, tagged unions, variant types, coproduct types)

Domain-level **Enums**

(a.k.a. sum types, tagged unions, variant types, coproduct types)

- Looking for domain-level analog of

```
enum Request:
  case ForOffice   (what: Equipment, desk: DeskLocation)
  case WorkFromHome(what: Equipment, addr: DeliveryAddress)
```

Domain-level Enums

(a.k.a. sum types, tagged unions, variant types, coproduct types)

- Looking for domain-level analog of

```
enum Request:
  case ForOffice   (what: Equipment, desk: DeskLocation)
  case WorkFromHome(what: Equipment, addr: DeliveryAddress)
```

- Why cannot use Scala enum?

Domain-level Enums

(a.k.a. sum types, tagged unions, variant types, coproduct types)

- Looking for domain-level analog of

```
enum Request:
  case ForOffice    (what: Equipment, desk: DeskLocation)
  case WorkFromHome(what: Equipment, addr: DeliveryAddress)
```

- Why cannot use Scala enum?

  ☢ to avoid *contamination* by Scala objects

Domain-level **Enums**

(a.k.a. sum types, tagged unions, variant types, coproduct types)

- Looking for domain-level analog of

```
enum Request:
  case ForOffice   (what: Equipment, desk: DeskLocation)
  case WorkFromHome(what: Equipment, addr: DeliveryAddress)
```

- Why cannot use Scala `enum`?
  - ☢ to avoid *contamination* by Scala objects
  - DEAD END would need *Scala* functions to work with

Domain-level **Enums**

(a.k.a. sum types, tagged unions, variant types, coproduct types)

- Looking for domain-level analog of

```
enum Request:
  case ForOffice   (what: Equipment, desk: DeskLocation)
  case WorkFromHome(what: Equipment, addr: DeliveryAddress)
```

- Why cannot use Scala enum?
  - ☢ to avoid *contamination* by Scala objects
  - DEAD END would need *Scala* functions to work with

- Aiming for

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

Domain-level **Enums**

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

Domain-level **Enums**

```
sealed trait **[A, B]
sealed trait Enum[Cases]
sealed trait ||[A, B]
sealed trait ::[Name, Type]
```

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

Domain-level **Enums**

```scala
sealed trait **[A, B]
sealed trait Enum[Cases]
sealed trait ||[A, B]
sealed trait ::[Name, Type]
```

```scala
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

What can we ***do*** with it?

Domain-level **Enums**

```scala
sealed trait **[A, B]
sealed trait Enum[Cases]
sealed trait ||[A, B]
sealed trait ::[Name, Type]
```

```scala
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

What do we *need* to ***do*** with it?

Domain-level **Enums**

```scala
sealed trait **[A, B]
sealed trait Enum[Cases]
sealed trait ||[A, B]
sealed trait ::[Name, Type]
```

```scala
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

What do we *need* to ***do*** with it?

Create Requests
from input data

```scala
Flow[Equipment ** DeskLocation,    Request]
Flow[Equipment ** DeliveryAddress, Request]
```

Domain-level **Enums**

```
sealed trait **[A, B]
sealed trait Enum[Cases]
sealed trait ||[A, B]
sealed trait ::[Name, Type]
```

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

What do we *need* to ***do*** with it?

Create Requests
from input data

```
Flow[Equipment ** DeskLocation,    Request]
Flow[Equipment ** DeliveryAddress, Request]
```

```
Flow[Equipment ** DeskLocation,    B]
Flow[Equipment ** DeliveryAddress, B]
```

by providing a handler for each case

⇒

Handle Requests

```
Flow[Request, B]
```

Domain-level **Enums**

What:

Create Requests

```
Flow[Equipment ** DeskLocation,    Request]
Flow[Equipment ** DeliveryAddress, Request]
```

Handle Requests

```
Flow[Equipment ** DeskLocation,    B]
Flow[Equipment ** DeliveryAddress, B]
```

$\Rightarrow$

```
Flow[Request, B]
```

Domain-level **Enums**

What:

Create Requests

```
Flow[Equipment ** DeskLocation,      Request]
Flow[Equipment ** DeliveryAddress, Request]
```

```
Flow[Equipment ** DeskLocation,      B]
Flow[Equipment ** DeliveryAddress, B]
```

⇒

Handle Requests

```
Flow[Request, B]
```

How: Capture the intent, in a type-safe manner.

Domain-level **Enums**

What:

Create Requests

```
Flow[Equipment ** DeskLocation,     Request]
Flow[Equipment ** DeliveryAddress, Request]
```

Handle Requests

```
Flow[Equipment ** DeskLocation,     B]
Flow[Equipment ** DeliveryAddress, B]
```

⇒

```
Flow[Request, B]
```

How: Capture the intent, in a type-safe manner.

```
// create
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
// handle
case class Handle[Cases, B]   (hs: Handlers[Cases, B])  extends Flow[Enum[Cases], B]
```

Producing **Enums**

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Request = Enum
  [ "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                                            : Flow[Equipment ** DeskLocation, Enum[Cases]]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                                         : Flow[Equipment ** DeliveryAddress, Enum[Cases]]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```scala
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```scala
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                           : Flow[Equipment ** DeskLocation, Enum[Cases]]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                           : Flow[Equipment ** DeliveryAddress, Enum[Cases]]
```

```scala
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                        : Flow[Equipment ** DeskLocation, Enum[Cases]]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                        : Flow[Equipment ** DeliveryAddress, Enum[Cases]]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                            : Flow[Equipment ** DeskLocation, Enum[Cases]]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                            : Flow[Equipment ** DeliveryAddress, Enum[Cases]]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                             : Flow[Equipment ** DeskLocation, Enum[Cases]]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                             : Flow[Equipment ** DeliveryAddress, Enum[Cases]]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                                   : Flow[Equipment ** DeskLocation,     Request]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                                 : Flow[Equipment ** DeliveryAddress,    Request]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

What we wanted

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                                      : Flow[Equipment ** DeskLocation,     Request]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                                   : Flow[Equipment ** DeliveryAddress,     Request]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                                    : Flow[Equipment ** DeskLocation,    Request]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                                 : Flow[Equipment ** DeliveryAddress,    Request]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Inject["ForOffice", Equipment ** DeskLocation, Cases](???)
                                : Flow[Equipment ** DeskLocation,     Request]
Inject["WorkFromHome", Equipment ** DeliveryAddress, Cases](???)
                             : Flow[Equipment ** DeliveryAddress,     Request]
```

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

Evidence that N :: A is one of Cases (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Member["ForOffice", Equipment ** DeskLocation,    Cases]
```

Evidence that N :: A is one of Cases (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Member[“ForOffice", Equipment ** DeskLocation,    Cases] ✔
```

Evidence that N :: A is one of **Cases** (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Member[“ForOffice", Equipment ** DeskLocation,    Cases] ✔
Member[“Foo",       Equipment ** DeskLocation,    Cases]
```

Evidence that **N :: A** is one of **Cases** (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Member["ForOffice", Equipment ** DeskLocation,    Cases] ✔
Member["Foo",       Equipment ** DeskLocation,    Cases] ✘
```

unrepresentable

Evidence that **N :: A** is one of **Cases** (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing **Enums**

```
type Cases =
  ( "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Member["ForOffice", Equipment ** DeskLocation,    Cases] ✔
Member["Foo",       Equipment ** DeskLocation,    Cases] ✘
Member["ForOffice", Equipment ** DeliveryAddress, Cases]
```

unrepresentable

Evidence that **N :: A** is one of **Cases** (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Producing Enums

```
type Cases =
  ( "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  )
type Request = Enum[Cases]
```

```
Member[“ForOffice", Equipment ** DeskLocation,     Cases] ✔
Member[“Foo",       Equipment ** DeskLocation,     Cases] ✘
Member[“ForOffice", Equipment ** DeliveryAddress, Cases] ✘
```

unrepresentable
unrepresentable

Evidence that **N :: A** is one of **Cases** (📗)

```
case class Inject[N, A, Cases](ev: Member[N, A, Cases]) extends Flow[A, Enum[Cases]]
```

Consuming Enums

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

Consuming Enums

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

collection of handlers, one for each case (📗)

Consuming Enums

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

collection of handlers, one for each case (📗)

```
type Cases = "x" :: Tuna || "y" :: Cod
```

Consuming Enums

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

collection of handlers, one for each case (📗)

```
type Cases = "x" :: Tuna || "y" :: Cod
```

```
Handle[Cases, B] =
```

```
Enum[ "x" :: Tuna || "y" :: Cod ]
```

B

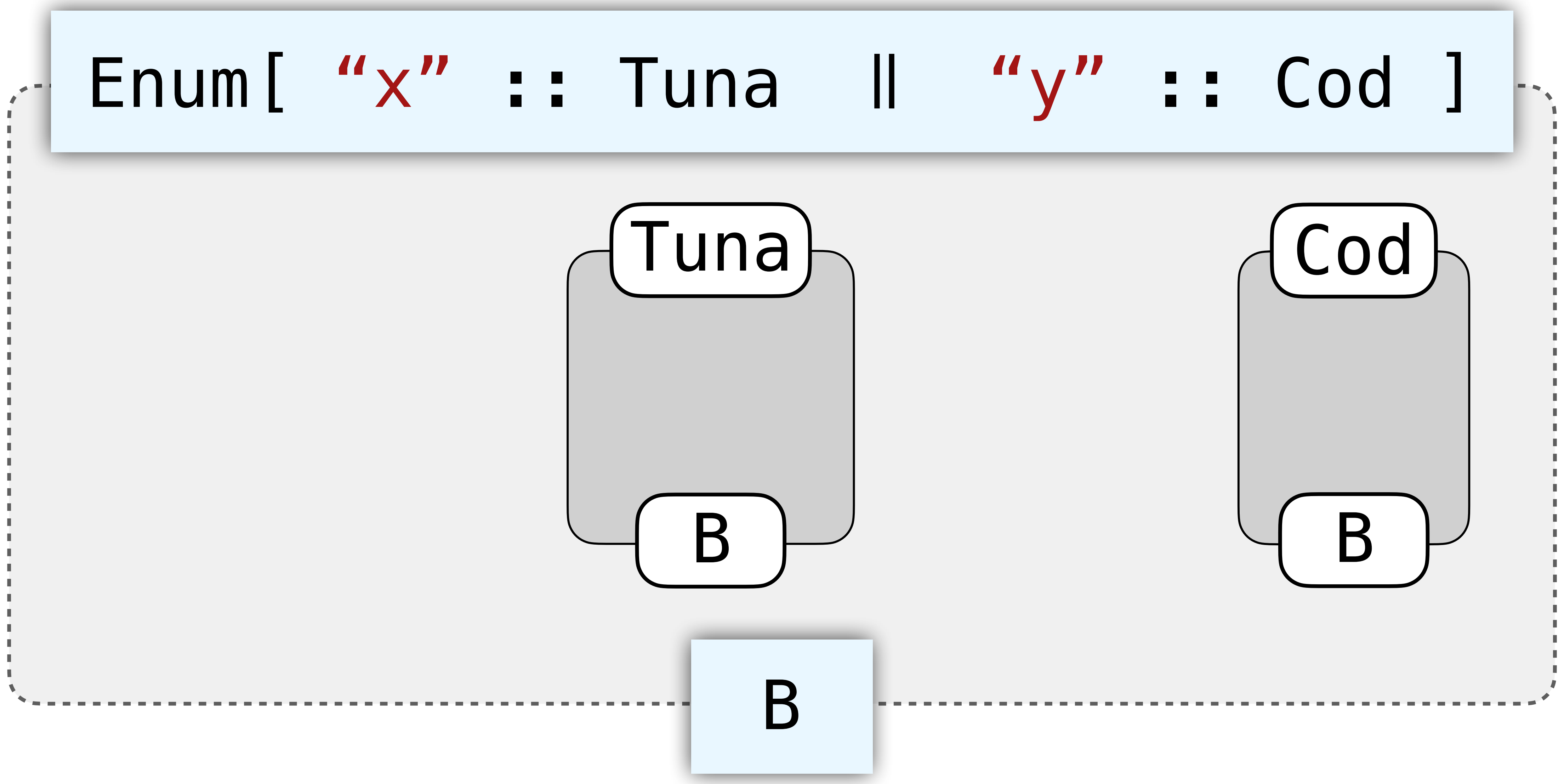
Consuming **Enums**

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

collection of handlers, one for each case (📗)

```
type Cases = “x” :: Tuna  ||  “y” :: Cod
```

Handle[Cases, B] =



Consuming **Enums**

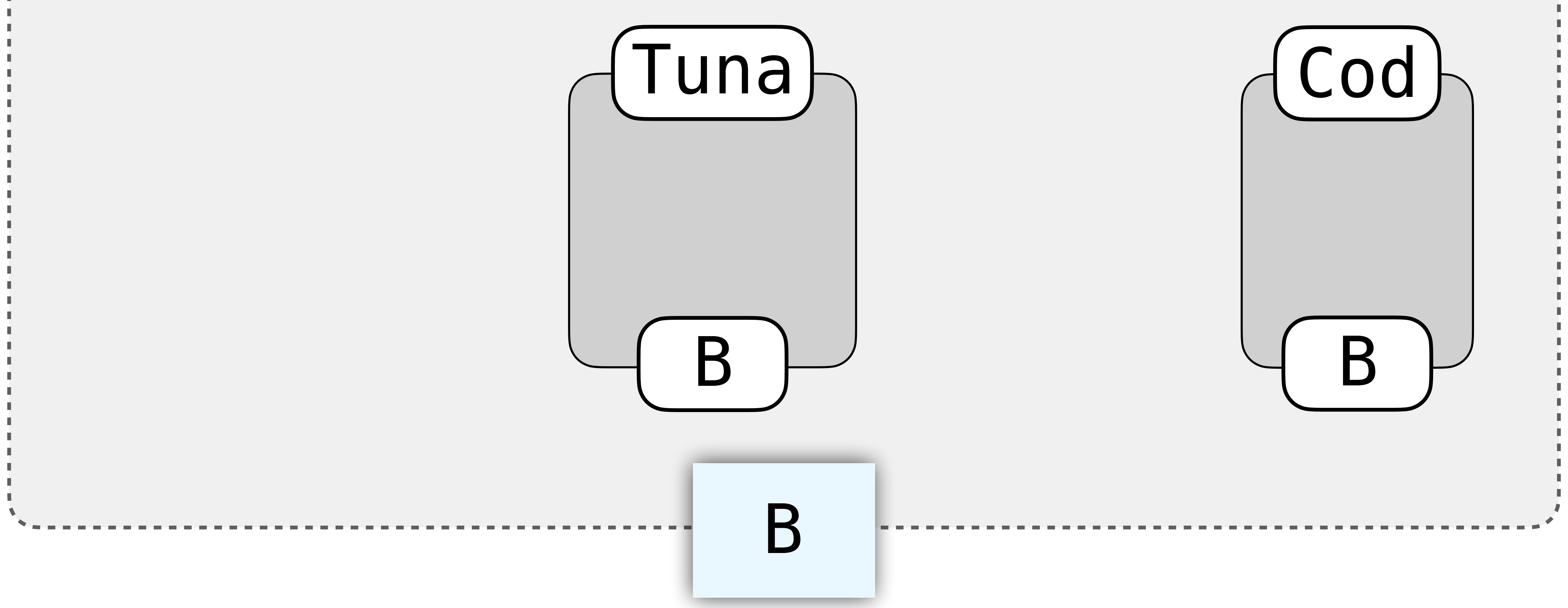
```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

collection of handlers, one for each case (📗)

```
type Cases = "x" :: Tuna || "y" :: Cod
```

```
Handle[Cases, B] =
```

```
Enum[ "x" :: Tuna || "y" :: Cod ]
```



non-exhaustive handlers *unrepresentable*

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Consuming **Enum** and a **Side Dish**

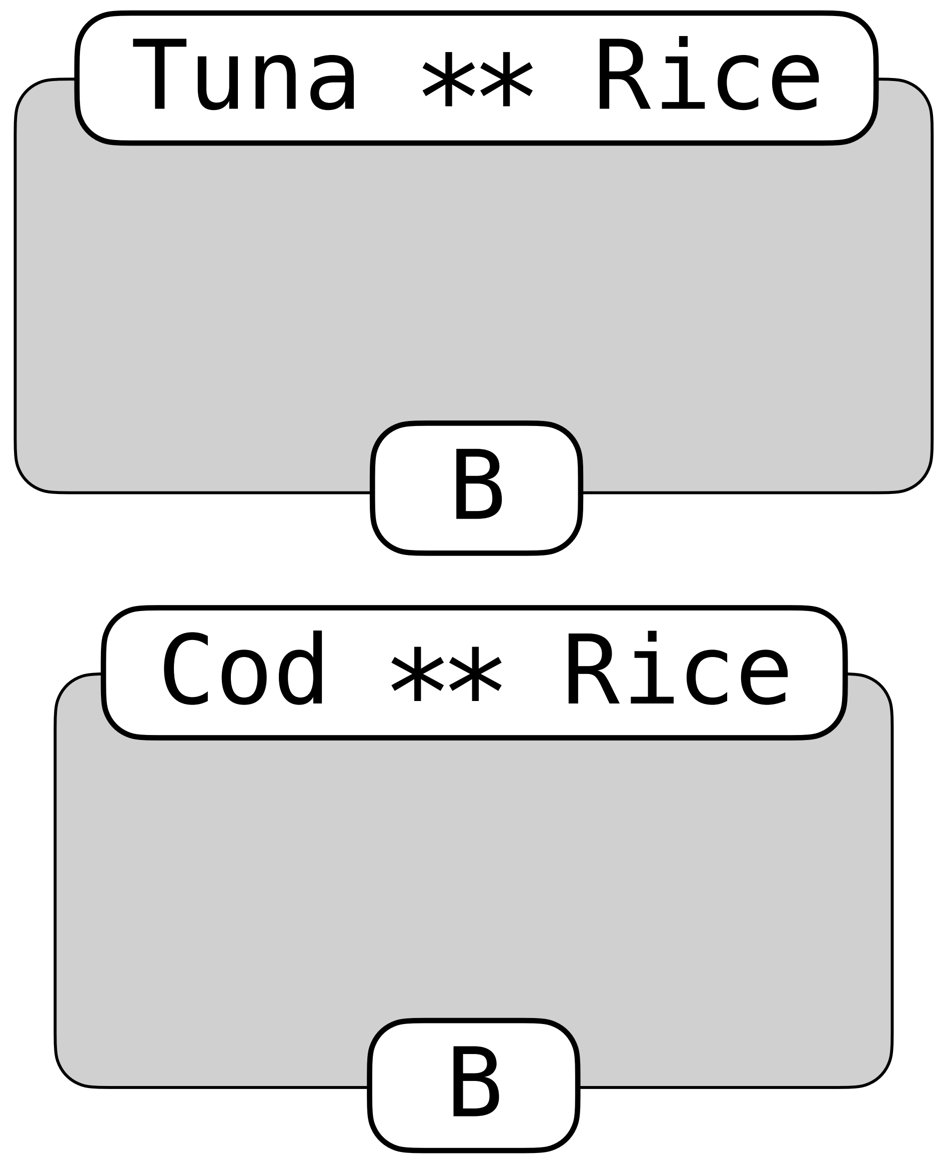
(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

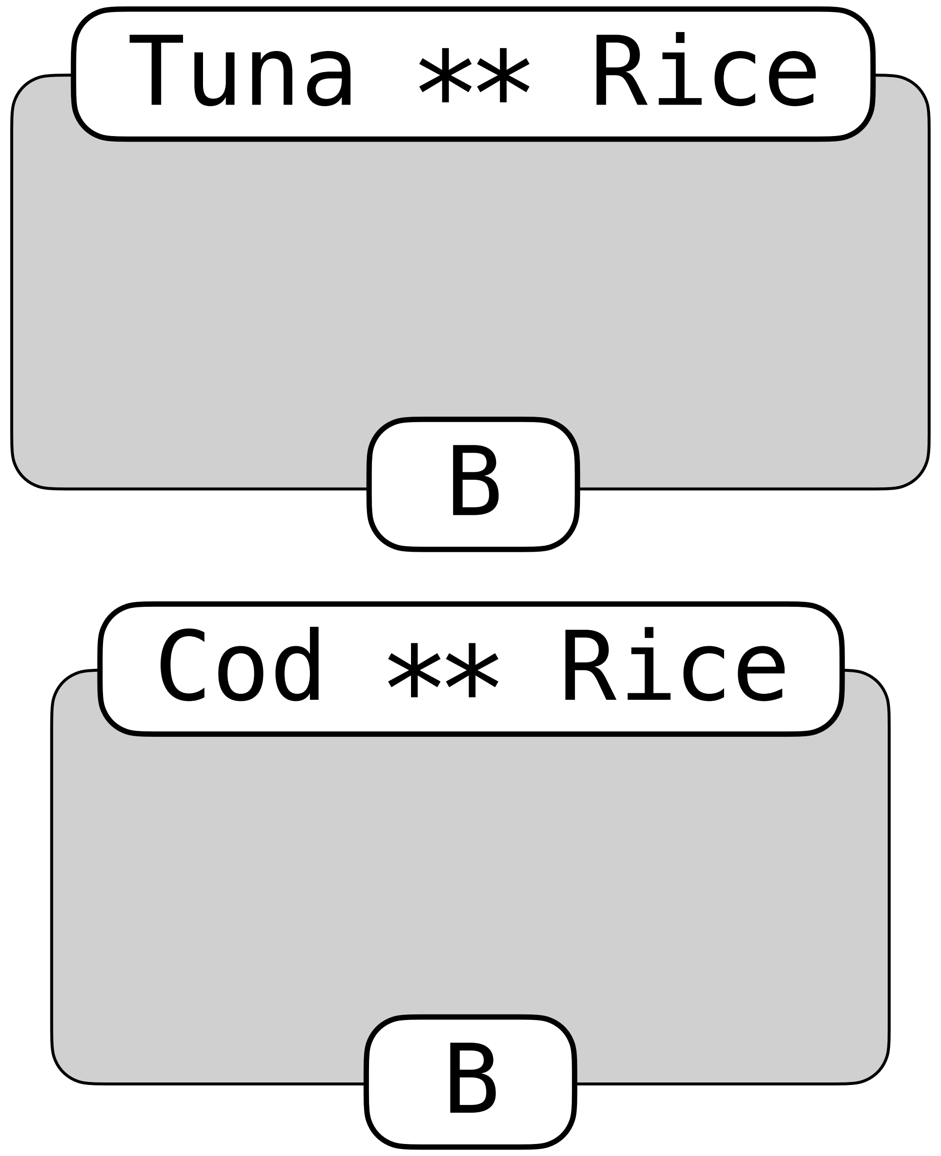


Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
Enum[ “x” :: Tuna  ||  “y” :: Cod ]       **       Rice
```



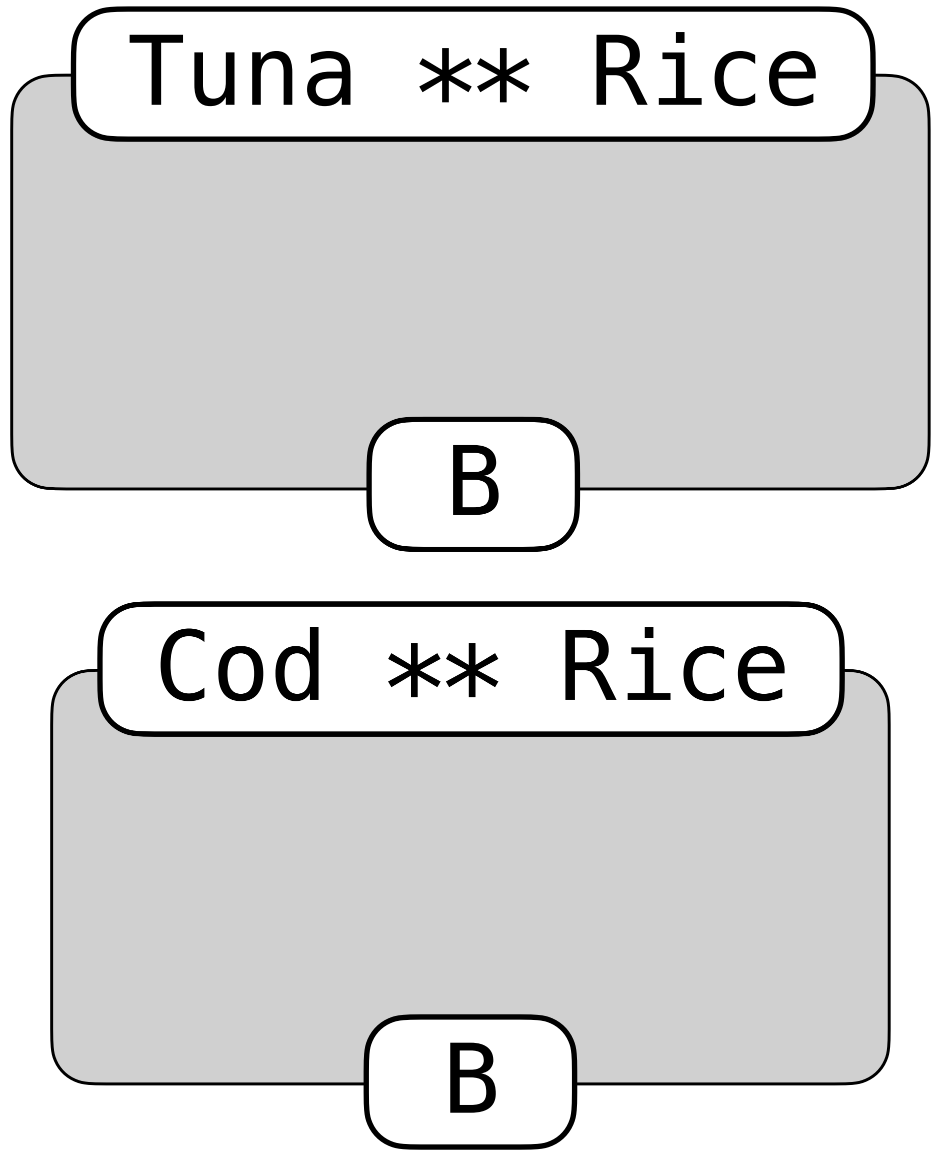
Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
Enum[ “x” :: Tuna  ||  “y” :: Cod ]        **        Rice
```

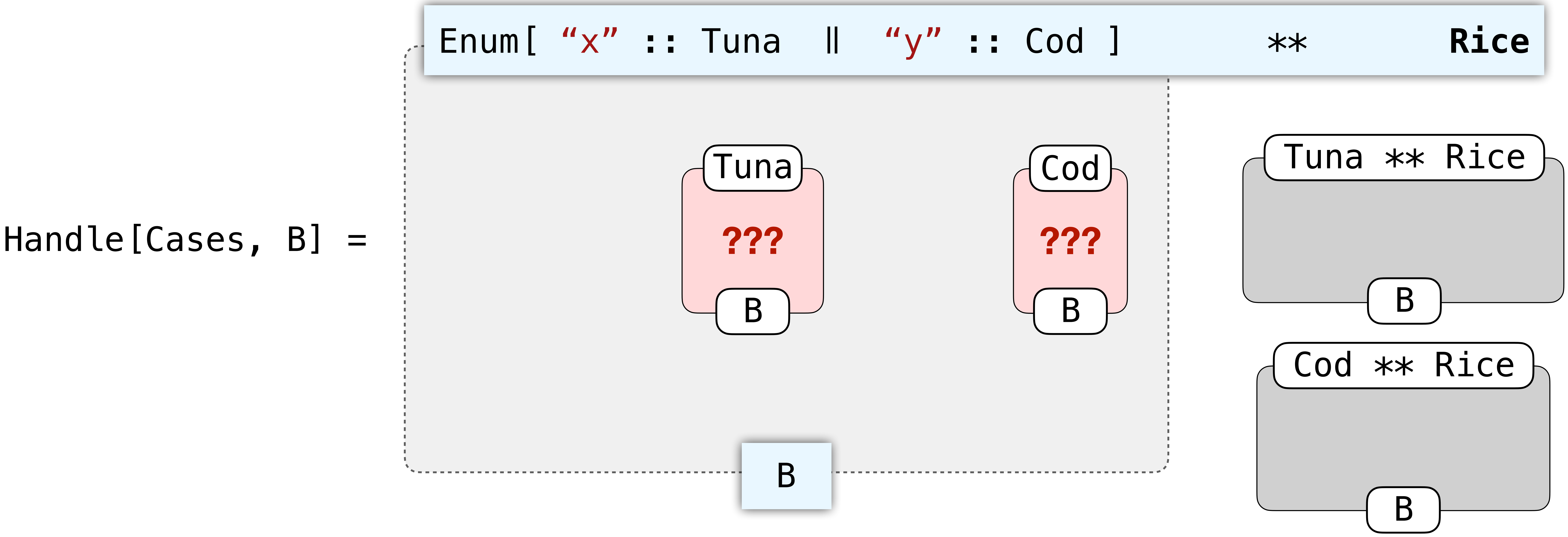


Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```



Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
Enum[ "x" :: Tuna  ||  "y" :: Cod ]       **       Rice
```

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```scala
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
Enum[ "x" :: Tuna  ||  "y" :: Cod ]        **        Rice
```

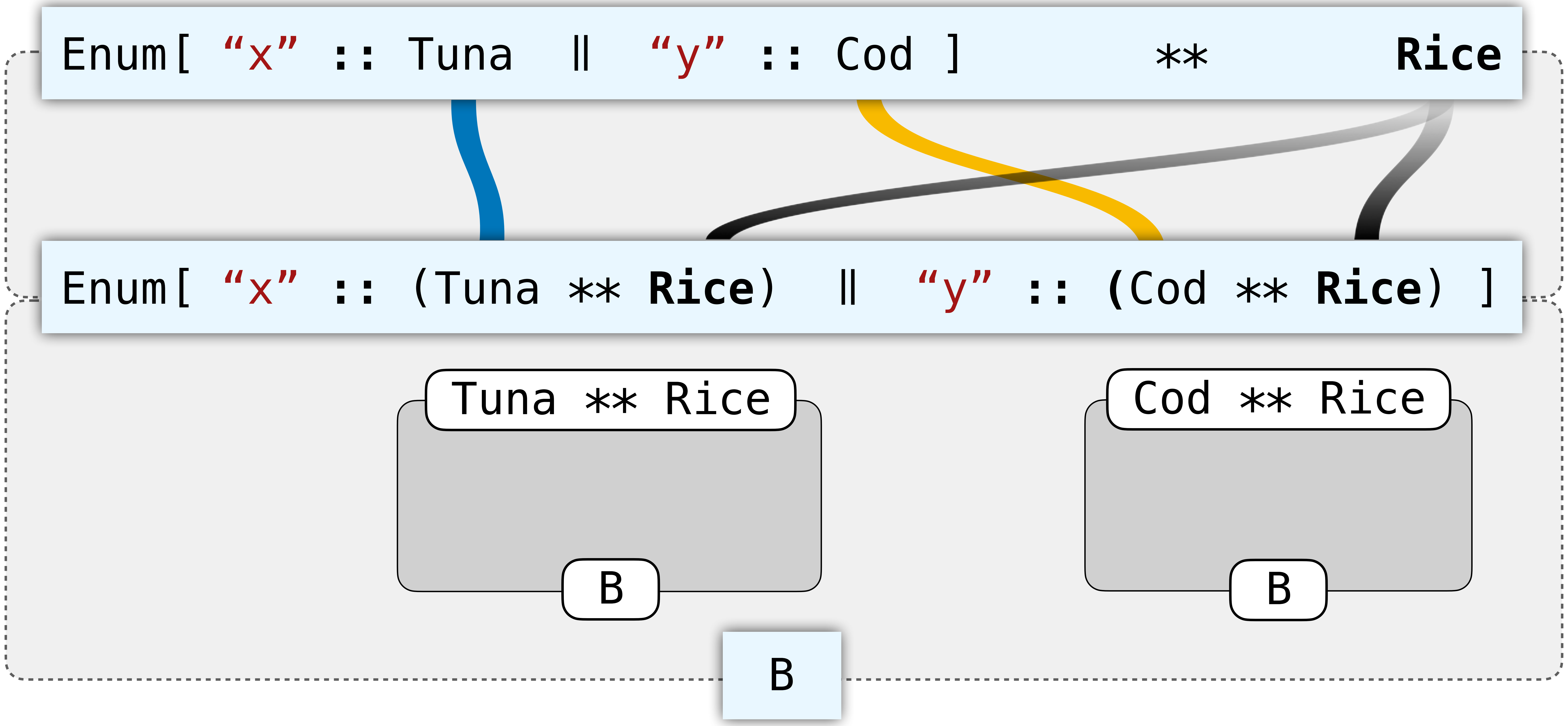
```
Enum[ "x" :: (Tuna ** Rice)  ||  "y" :: (Cod ** Rice) ]
```

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```



Consuming **Enum** and a **Side Dish**

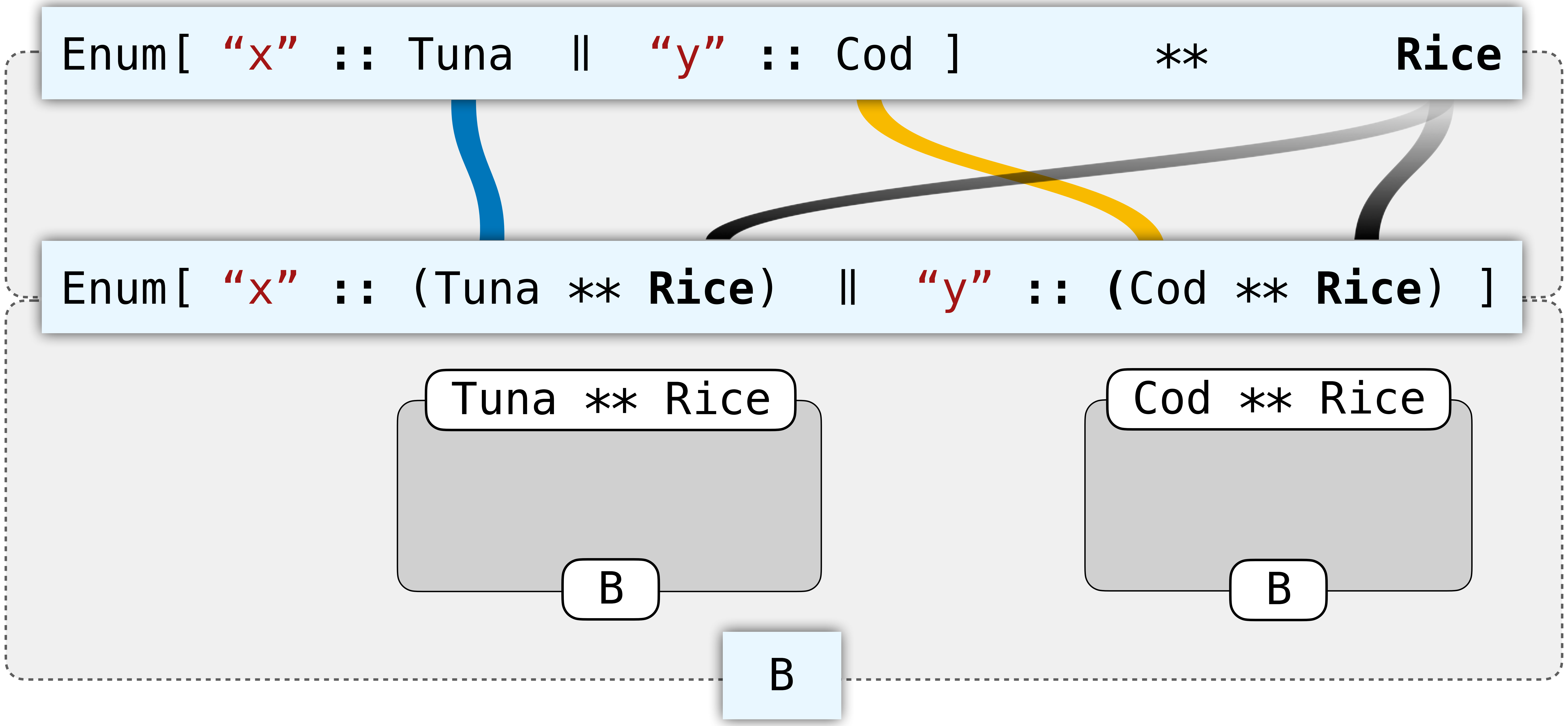
(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
DistributeRL[
  Rice,
  Cases,
  Cases¹
]
```

=



Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
DistributeRL[
  Rice,
  Cases,          =   Enum[ "x" :: Tuna  ||  "y" :: Cod ]       **        Rice
  Cases¹
]                     Enum[ "x" :: (Tuna ** Rice)  ||  "y" :: (Cod ** Rice) ]
```

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
DistributeRL[
  Rice,
  Cases,
  Cases¹
]
          =    Enum[ "x" :: Tuna  ‖  "y" :: Cod ]      **      Rice

               Enum[ "x" :: (Tuna ** Rice)  ‖  "y" :: (Cod ** Rice) ]
```

- **idea: just capture the intent**

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
DistributeRL[
  Rice,
  Cases,
  Cases¹
]                 =   Enum[ “x” :: Tuna  ‖  “y” :: Cod ]      **      Rice

                      Enum[ “x” :: (Tuna ** Rice)  ‖  “y” :: (Cod ** Rice) ]
```

```
case class DistributeRL[R, Cases, Cases¹](
  [ ... evidence that Cases¹ is the result
        of distributing R into Cases ... ]
) extends Flow[Enum[Cases] ** R, Enum[Cases¹]]
```

- **idea: just capture the intent**

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

```
DistributeRL[
  Rice,
  Cases,
  Cases¹
]                =

Enum[ “x” :: Tuna  ||  “y” :: Cod ]        **        Rice

Enum[ “x” :: (Tuna ** Rice)  ||  “y” :: (Cod ** Rice) ]
```

```
case class DistributeRL[R, Cases, Cases¹](
  [ ... evidence that Cases¹ is the result
        of distributing R into Cases ... ]
) extends Flow[Enum[Cases] ** R, Enum[Cases¹]]
```

- **idea: just capture the intent**
- completely type-safe

Consuming **Enum** and a **Side Dish**

(needed to support pattern matching with **capture**)

Problem: What if I cannot consume Tuna/Cod without Rice

```
case class Handle[Cases, B](hs: Handlers[Cases, B]) extends Flow[Enum[Cases], B]
```

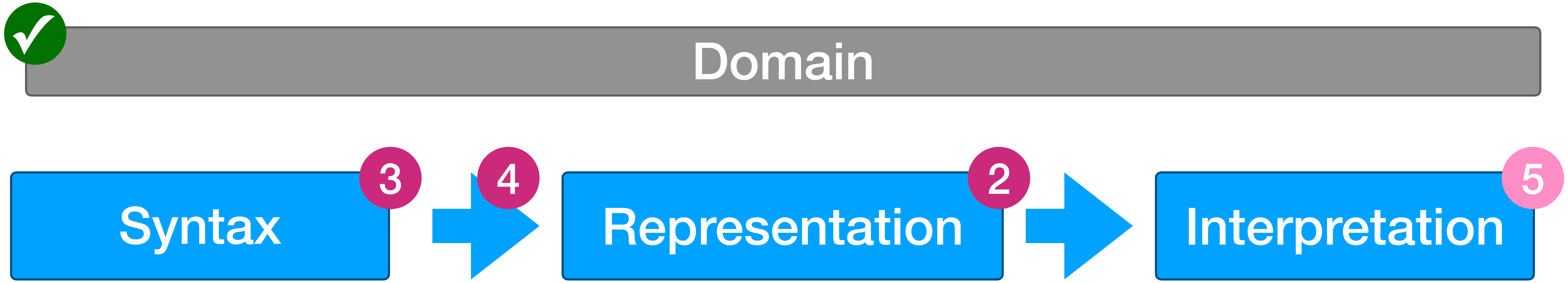
```
DistributeRL[
  Rice,
  Cases,
  Cases¹
]                =   Enum[ “x” :: Tuna  ||  “y” :: Cod ]        **        Rice

                     Enum[ “x” :: (Tuna ** Rice)  ||  “y” :: (Cod ** Rice) ]
```

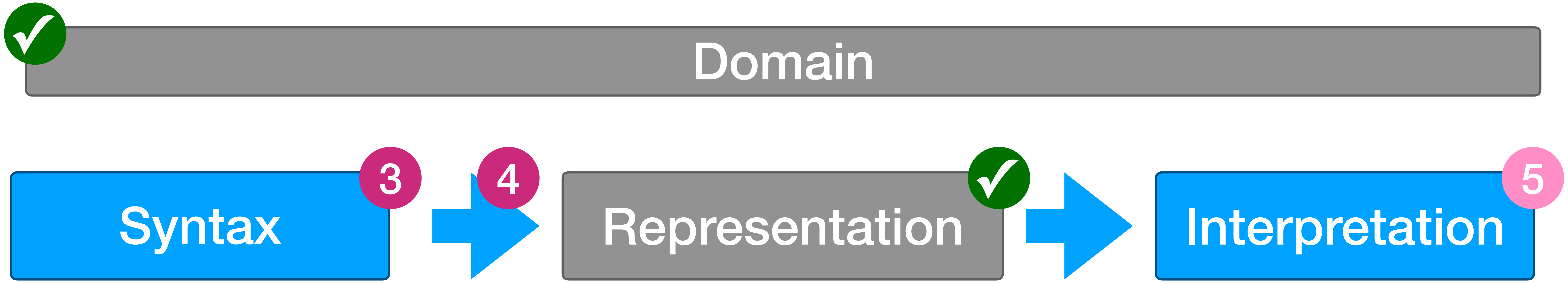
```
case class DistributeRL[R, Cases, Cases¹](
  [ ... evidence that Cases¹ is the result
        of distributing R into Cases ... ]
) extends Flow[Enum[Cases] ** R, Enum[Cases¹]]
```

- **idea: just capture the intent**
- completely type-safe
- works for any number of cases

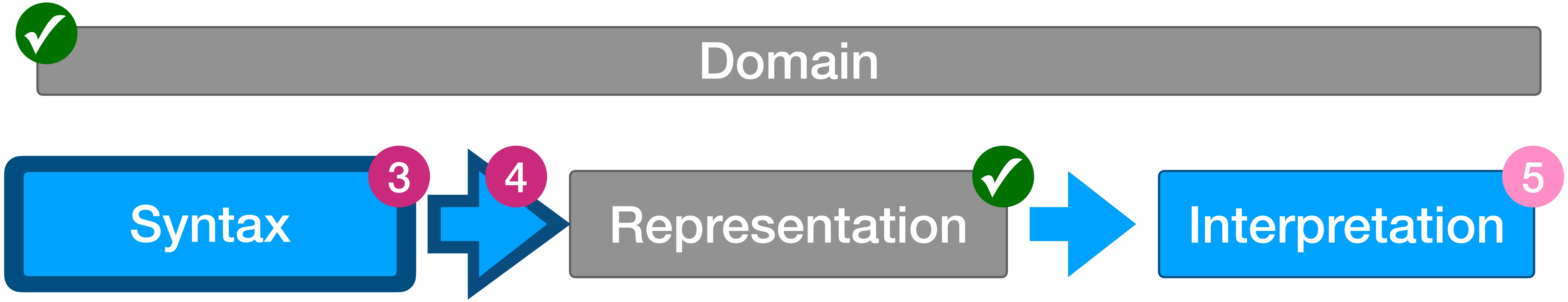
Agenda



Agenda



Agenda



Why any special Syntax?

Why not construct instances of `Flow[A,B]` directly?

Why any special Syntax?

Why not construct instances of `Flow[A,B]` directly?

```
AndThen(
  Peel(),
  Either(
    AndThen(
      Extract(),
      AndThen(
        AndThen(
          AndThen(
            Par(Peel(), Id()),
            AndThen(
              AndThen(Swap(), DistributeLR()),
              Either(
                AndThen(Swap(), InjectL()),
                AndThen(Swap(), InjectR())
              )
            )
          ),
          Either(
            AndThen(
              AndThen(
                AndThen(Par(Extract(), Id()), Inject(Single(Monitor))),
                InjectL()
              ),
              Unpeel()
            ),
            Inject(InLast(Chair))
          )
        ),
        AndThen(
          Peel(),
          Either(
            AndThen(
              Extract(),
              AndThen(
                Par(Id(), Ext(RequestMonitorFromIT)),
                Prj2()
              )
            ),
            AndThen(
              Par(
                Id(),
                Ext(RequestChairFromOfficeMgmt)
              ),
              Prj2()
            )
          )
        )
      )
    ),
    Ext(OrderFromSupplier)
  )
)
```

Why any special Syntax?

Why not construct instances of `Flow[A,B]` directly?

```
AndThen(
  Peel(),
  Either(
    AndThen(
      Extract(),
      AndThen(
        AndThen(
          AndThen(
            Par(Peel(), Id()),
            AndThen(
              AndThen(Swap(), DistributeLR()),
              Either(
                AndThen(Swap(), InjectL()),
                AndThen(Swap(), InjectR())
              )
            )
          ),
          Either(
            AndThen(
              AndThen(
                AndThen(Par(Extract(), Id()), Inject(Single(Monitor))),
                InjectL()
              ),
              Unpeel()
            ),
            Inject(InLast(Chair))
          )
        ),
        AndThen(
          Peel(),
          Either(
            AndThen(
              Extract(),
              AndThen(
                Par(Id(), Ext(RequestMonitorFromIT)),
                Prj2()
              )
            ),
            AndThen(
              Par(
                Id(),
                Ext(RequestChairFromOfficeMgmt)
              ),
              Prj2()
            )
          )
        )
      )
    ),
    Ext(OrderFromSupplier)
  )
)
```

Can we get back variables and expressions?

Desired Syntax

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

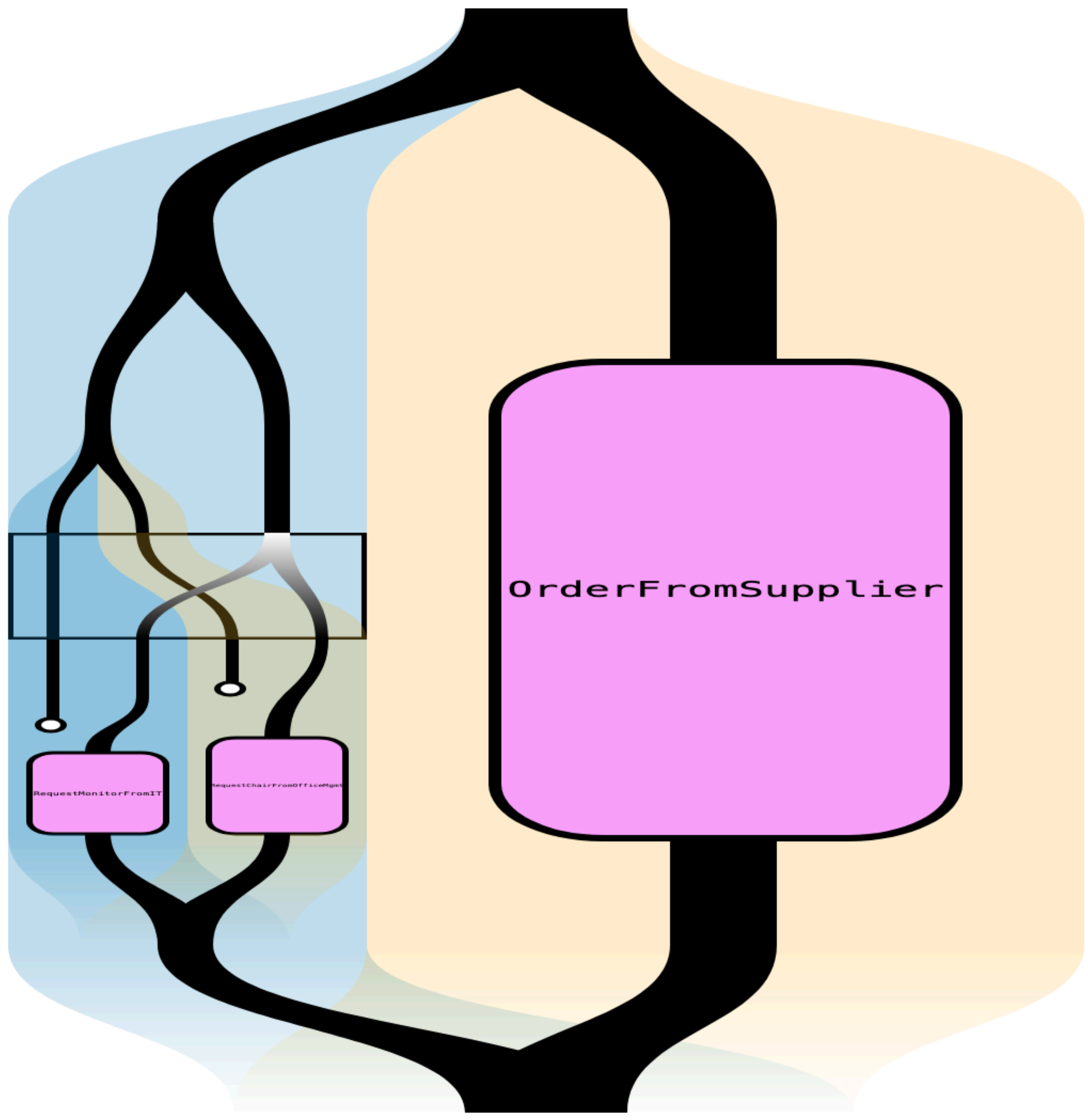
- Lambdas
- Variables
- Pattern-matching
- Expressions

Desired Syntax

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

- Lambdas
- Variables
- Pattern-matching
- Expressions

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```



Let’s Break It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

Let's Break It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do?

Let’s Break It Down

```
Flow { req =>                                              🧑‍💻
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do?
2. What does **`switch`** do?

Let's Break It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do?
2. What does **`switch`** do?
3. What do the **extractors** do? (`ForOffice`, `Monitor`, ...)

Let’s Break It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? 👈
2. What does **`switch`** do?
3. What do the **extractors** do? (`ForOffice`, `Monitor`, …)

Flow *"Compiles"* Scala Functions

```
Flow { req =>

  ???

}
```

Flow *"Compiles"* Scala Functions

```
Flow { req =>

  ???

}
```

- Takes a *Scala* function

Flow *"Compiles"* Scala Functions

```scala
Flow { req =>

  ???

} : Flow[Request, Result]
```

- Takes a *Scala* function
- Returns a `Flow`

Flow *“Compiles”* Scala Functions

```
Flow { req =>

  ???

} : Flow[Request, Result]
```

- Takes a *Scala* function
- Returns a `Flow`
  - **without** Scala functions

Flow *“Compiles”* Scala Functions

```
Flow { (req: Expr[Request]) => 👨‍💻

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**

- Returns a `Flow`
  - **without** Scala functions

Flow *"Compiles"* Scala Functions

```scala
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions

```scala
object Flow:




  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    ???
```

Flow *“Compiles”* Scala Functions

```scala
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions
- **Uses a library!** (`libretto-lambda`)

```scala
object Flow:






  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    ???
```

Flow *"Compiles"* Scala Functions

```scala
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions
- **Uses a library!** (`libretto-lambda`)

```scala
object Flow:



  opaque type Expr[A] = lambdas.Expr[A]

  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    ???
```

Flow *“Compiles”* Scala Functions

```
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions
- **Uses a library!** (`libretto-lambda`)

```
import libretto.lambda.Lambdas

object Flow:

  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)

  opaque type Expr[A] = lambdas.Expr[A]

  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    ???
```

Flow *"Compiles"* Scala Functions

```scala
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions
- **Uses a library!** (`libretto-lambda`)

```scala
import libretto.lambda.Lambdas

object Flow:

  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)

  opaque type Expr[A] = lambdas.Expr[A]

  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
    // ... and handle errors ...
```

Flow *"Compiles"* Scala Functions

```scala
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions
- **Uses a library!** (`libretto-lambda`)

```scala
import libretto.lambda.Lambdas

object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)

  opaque type Expr[A] = lambdas.Expr[A]

  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
    // ... and handle errors ...
```

Flow *"Compiles"* Scala Functions

```scala
Flow { (req: Expr[Request]) =>

  ??? : Expr[Result]

} : Flow[Request, Result]
```

- Takes a *Scala* function
  - **on auxiliary expressions**
- Returns a `Flow`
  - **without** Scala functions
- **Uses a library!** (`libretto-lambda`)

```scala
import libretto.lambda.Lambdas

object Flow:
  val lambdas: Lambdas[Flow, **, …] =
    Lambdas[Flow, **, …](...)

  opaque type Expr[A] = lambdas.Expr[A]

  def apply[A, B](
    f: Expr[A] => Expr[B],
  ): Flow[A, B] =
    lambdas.delambdify(..., f)
    // ... and handle errors ...
```

Peek into `libretto-lambda`

Peek into `libretto-lambda`

```
enum Expr[A]: // approximately
```

📗

Peek into `libretto-lambda`

```
enum Expr[A]: // approximately
  case Var(id: Object)
```

- variables

Peek into `libretto-lambda`

```
enum Expr[A]: // approximately
  case Var(id: Object)
  case Zip[A, B](
    a: Expr[A],
    b: Expr[B]
  ) extends Expr[A ** B]
```

- variables
- forming pairs

Peek into `libretto-lambda`

```
enum Expr[A]: // approximately
  case Var(id: Object)
  case Zip[A, B](
    a: Expr[A],
    b: Expr[B]
  ) extends Expr[A ** B]

  case Prj1[A, B](e: Expr[A ** B]) extends Expr[A]
  case Prj2[A, B](e: Expr[A ** B]) extends Expr[B]
```

- variables
- forming pairs
- accessing pairs

Peek into `libretto-lambda`

```scala
enum Expr[A]: // approximately
  case Var(id: Object)
  case Zip[A, B](
    a: Expr[A],
    b: Expr[B]
  ) extends Expr[A ** B]

  case Prj1[A, B](e: Expr[A ** B]) extends Expr[A]
  case Prj2[A, B](e: Expr[A ** B]) extends Expr[B]
  case Map[A, B](
    a: Expr[A],
    f: Flow[A, B]
  ) extends Expr[B]
```

- variables
- forming pairs
- accessing pairs
- applying an *already compiled* `Flow`

Peek into `libretto-lambda`

```
enum Expr[A]: // approximately
  case Var(id: Object)
  case Zip[A, B](
    a: Expr[A],
    b: Expr[B]
  ) extends Expr[A ** B]

  case Prj1[A, B](e: Expr[A ** B]) extends Expr[A]
  case Prj2[A, B](e: Expr[A ** B]) extends Expr[B]
  case Map[A, B](
    a: Expr[A],
    f: Flow[A, B]
  ) extends Expr[B]
```

- variables
- forming pairs
- accessing pairs
- applying an *already compiled* `Flow`
- **no** lambda abstraction (immediately delambdified)

Peek into libretto-lambda

```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
```

Peek into `libretto-lambda`

```
// approximately 📗
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
```

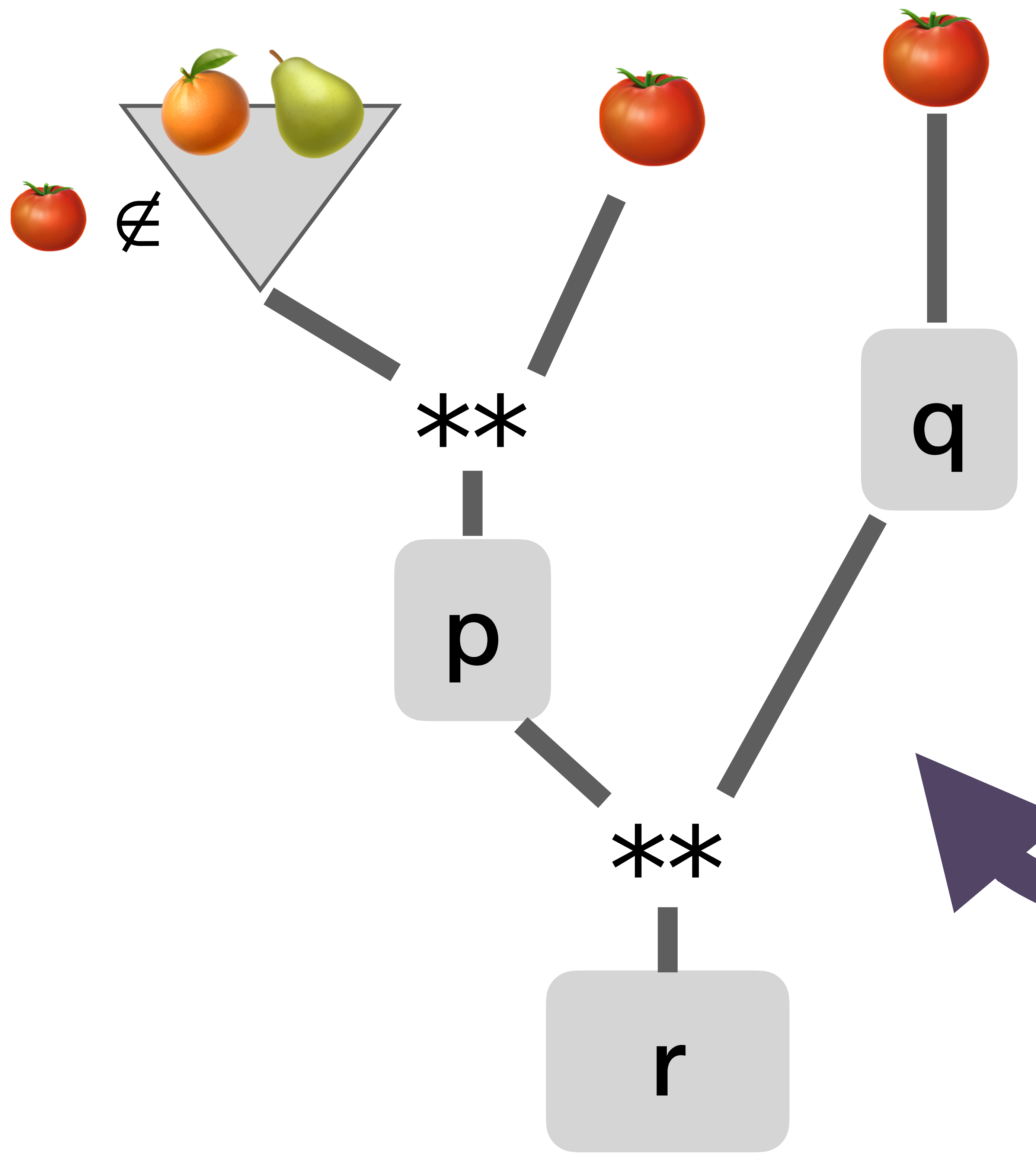
Peek into `libretto-lambda`

```scala
// approximately                                      📗
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
```

Peek into `libretto-lambda`

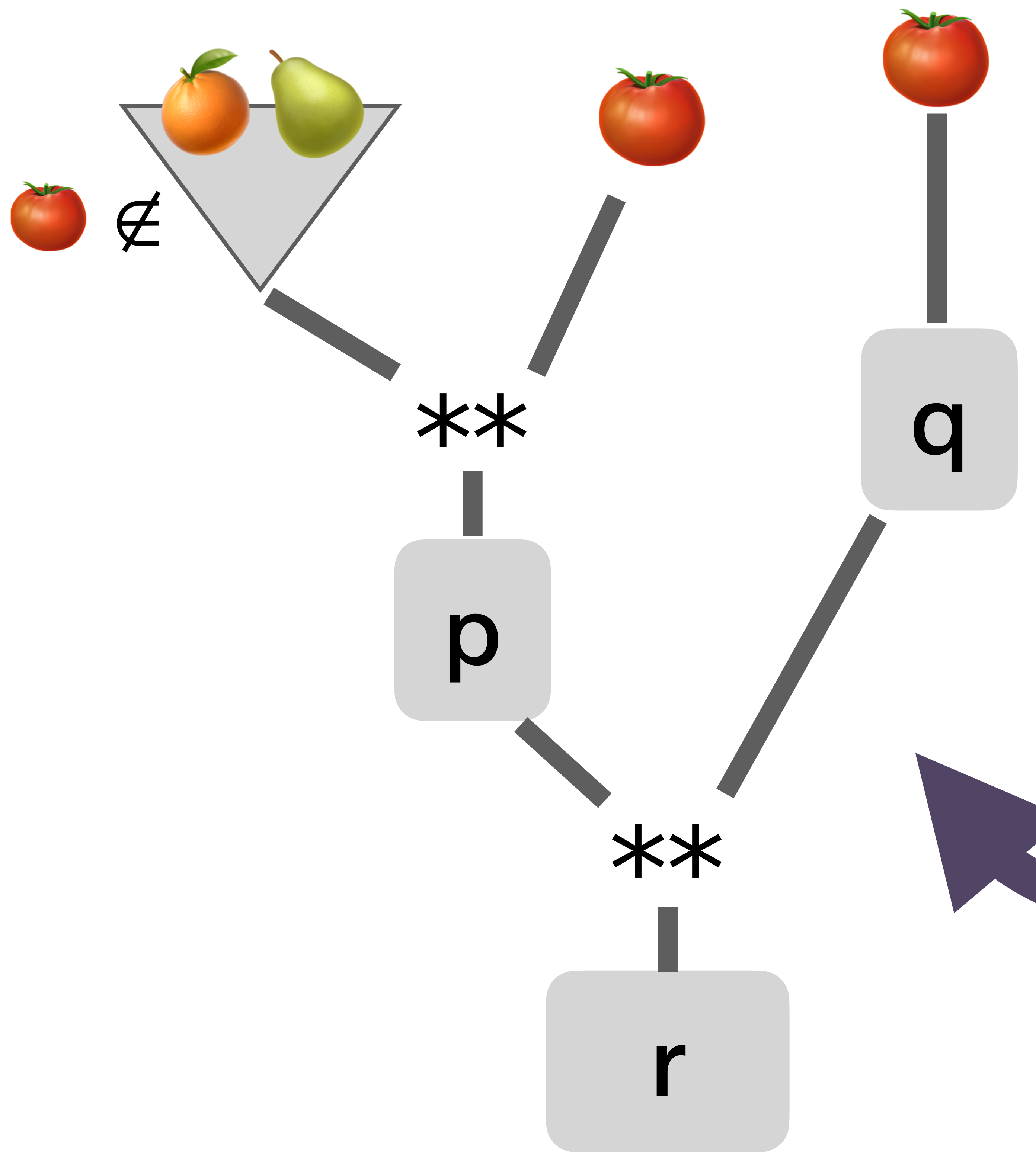
```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
```

Peek into `libretto-lambda`



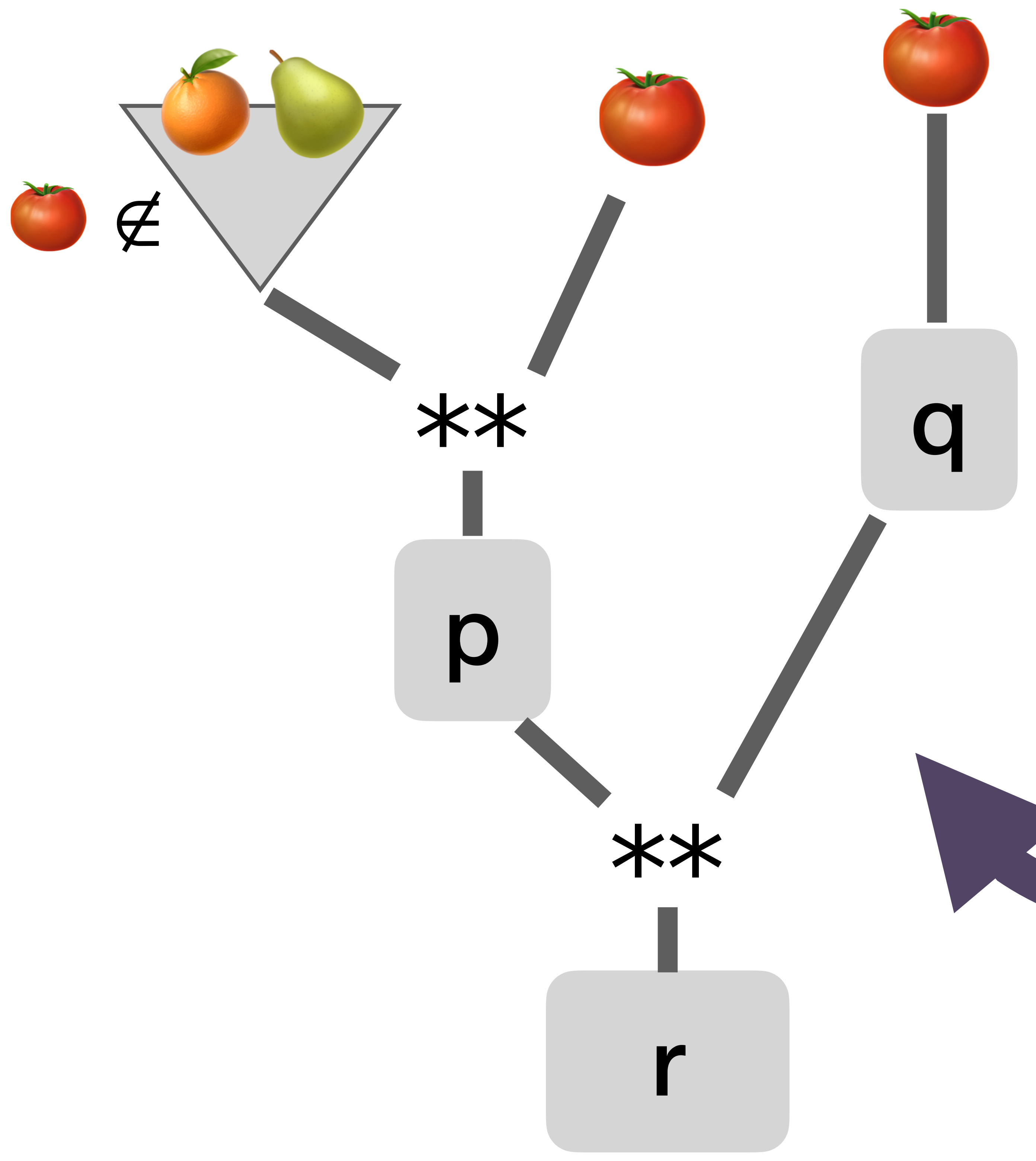

```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
```

Peek into `libretto-lambda`



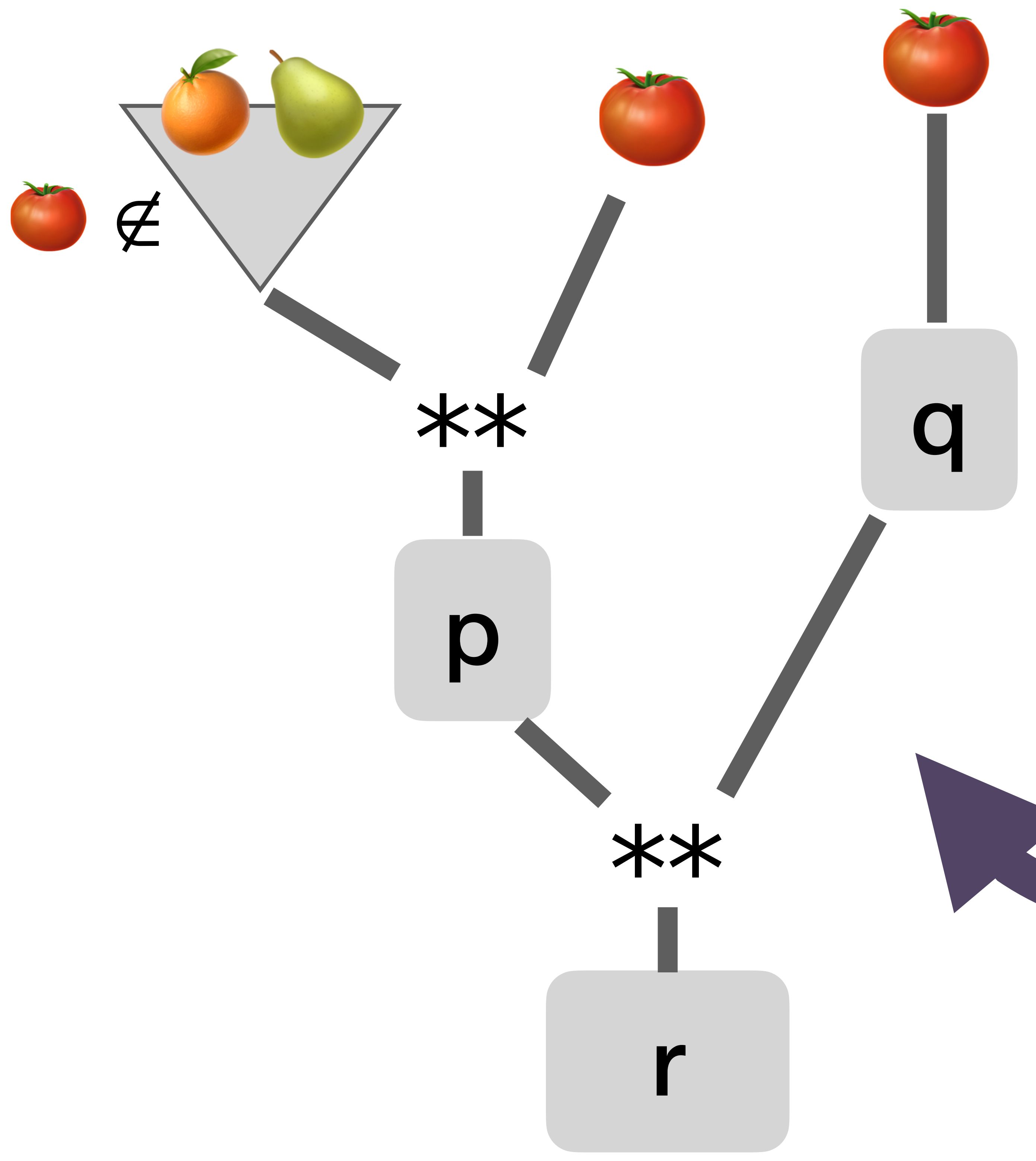

```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
  eliminate(🍅, from = b)
```

Peek into `libretto-lambda`

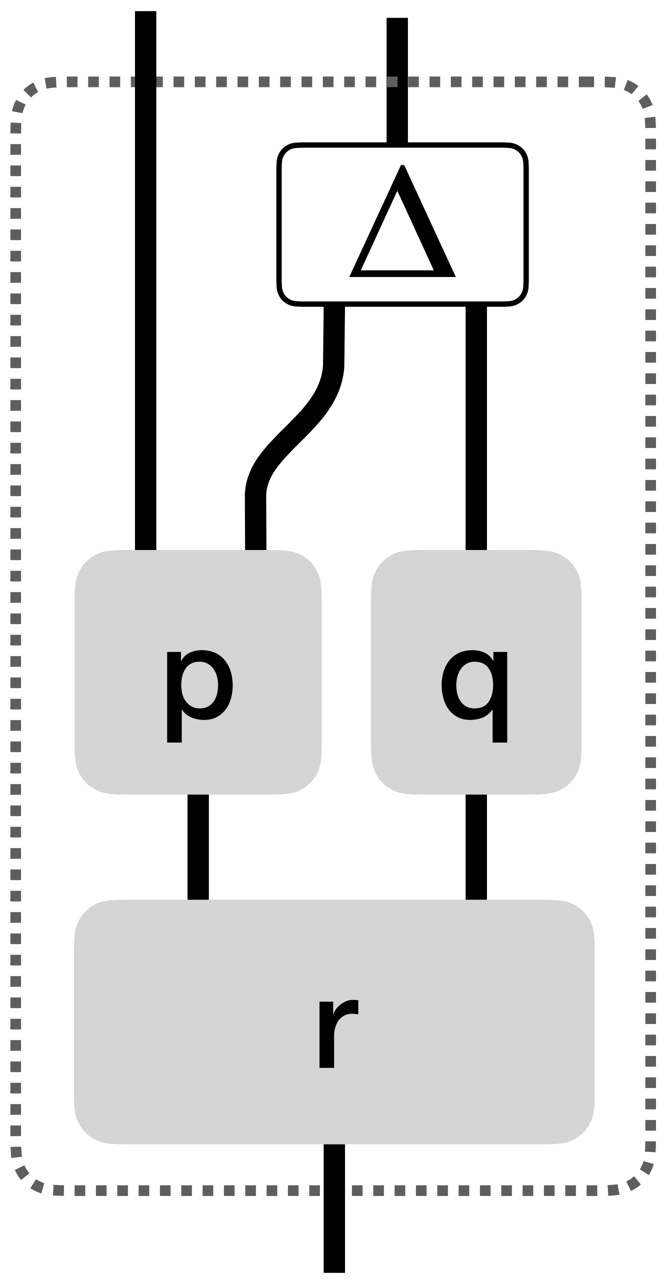



```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
  eliminate(🍅, from = b)
```

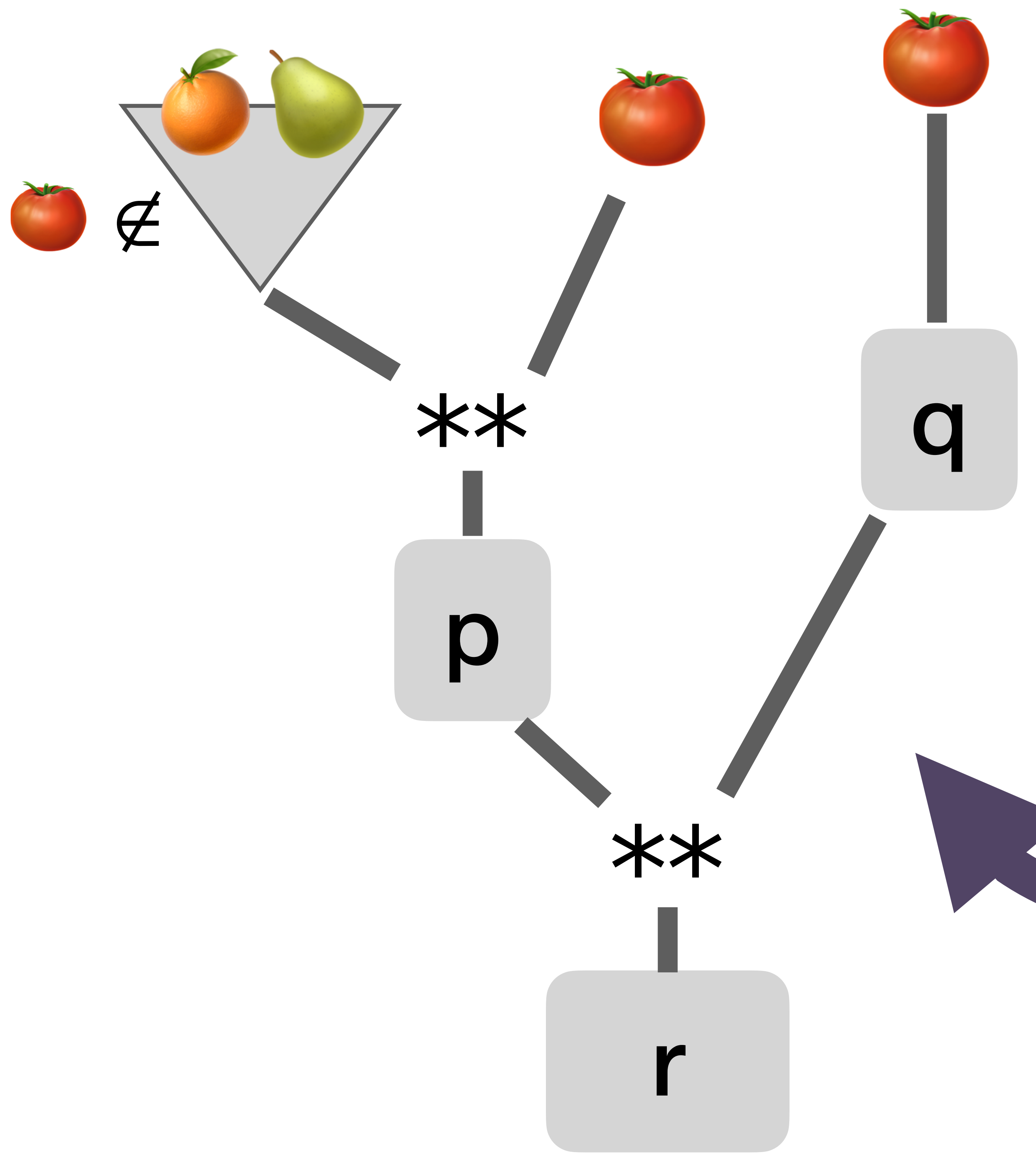
Peek into `libretto-lambda`



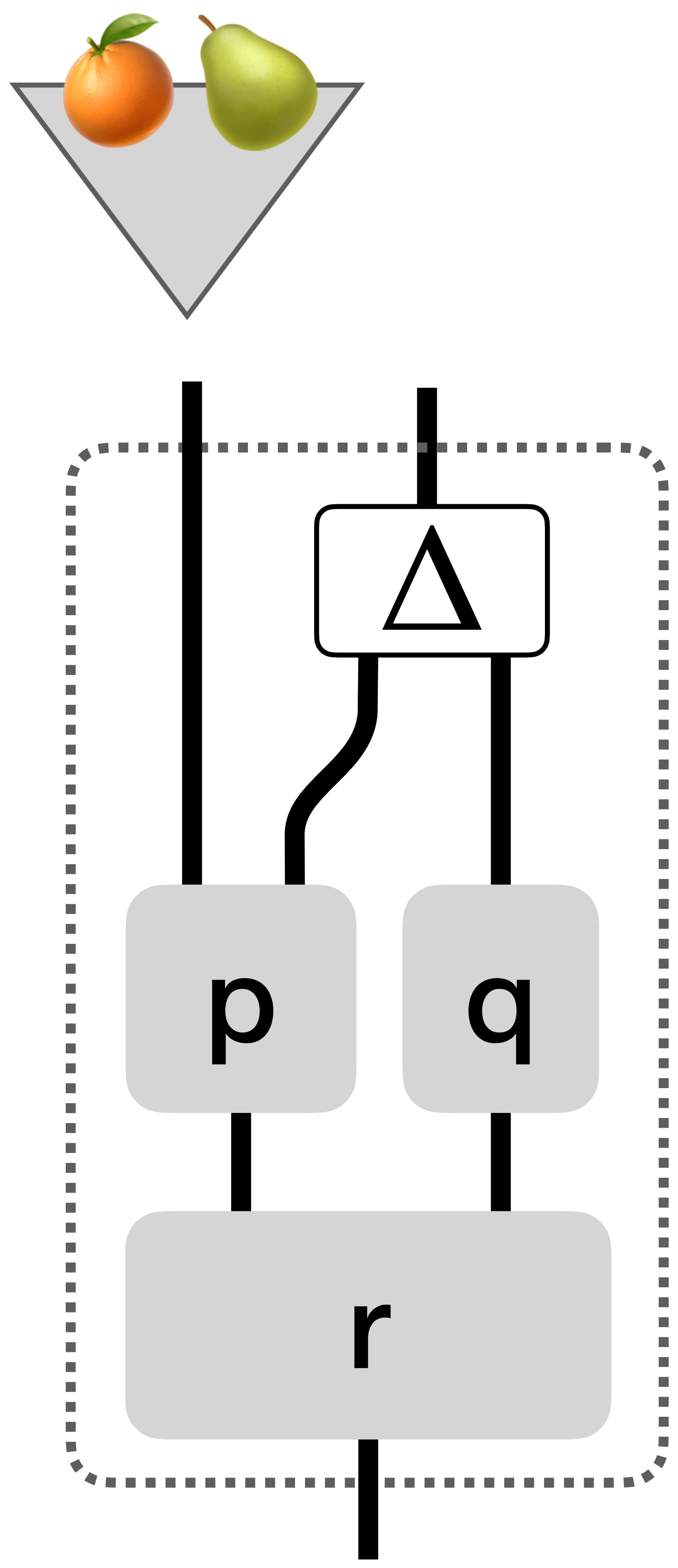

```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
  eliminate(🍅, from = b)
```



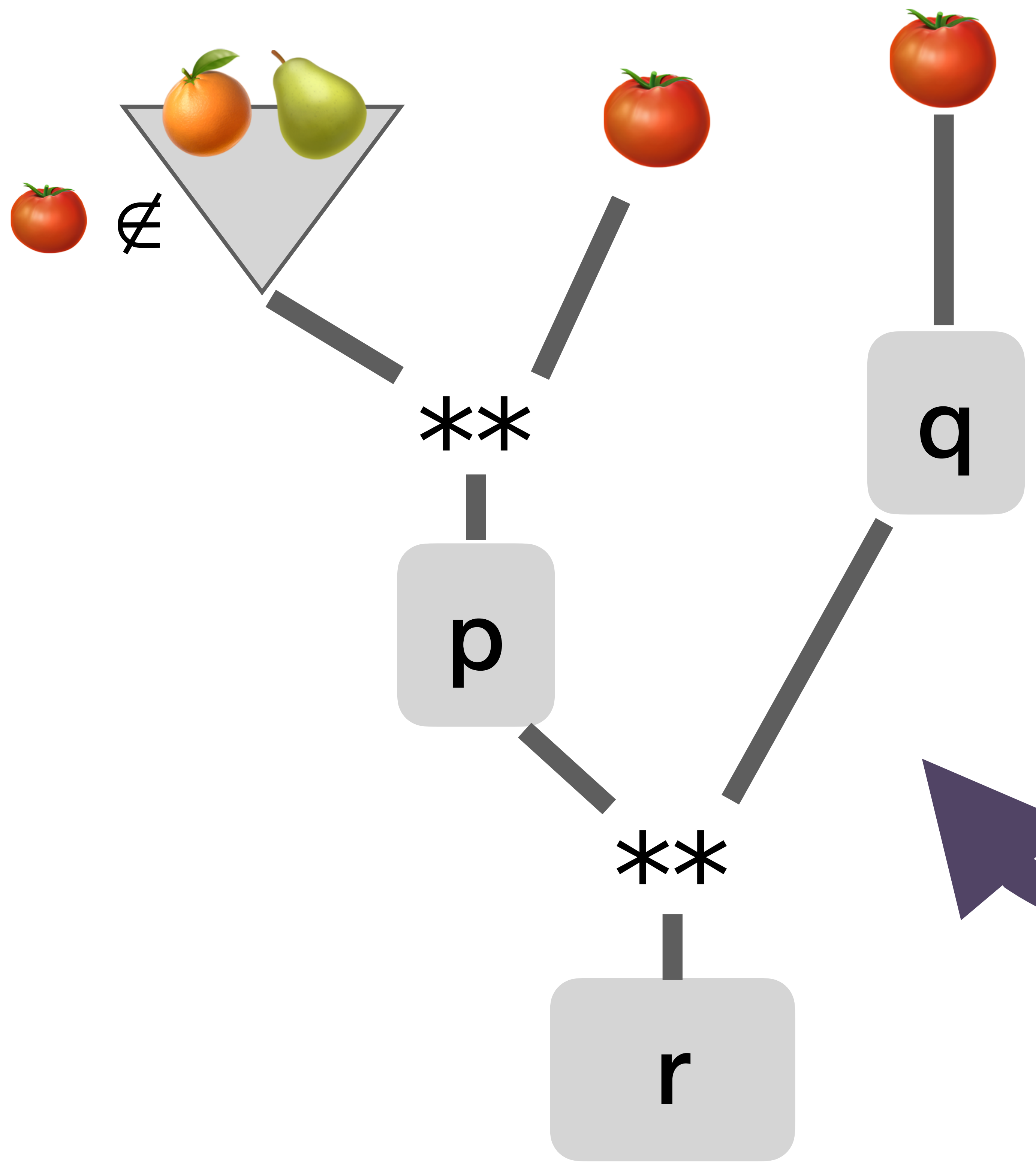
Peek into `libretto-lambda`



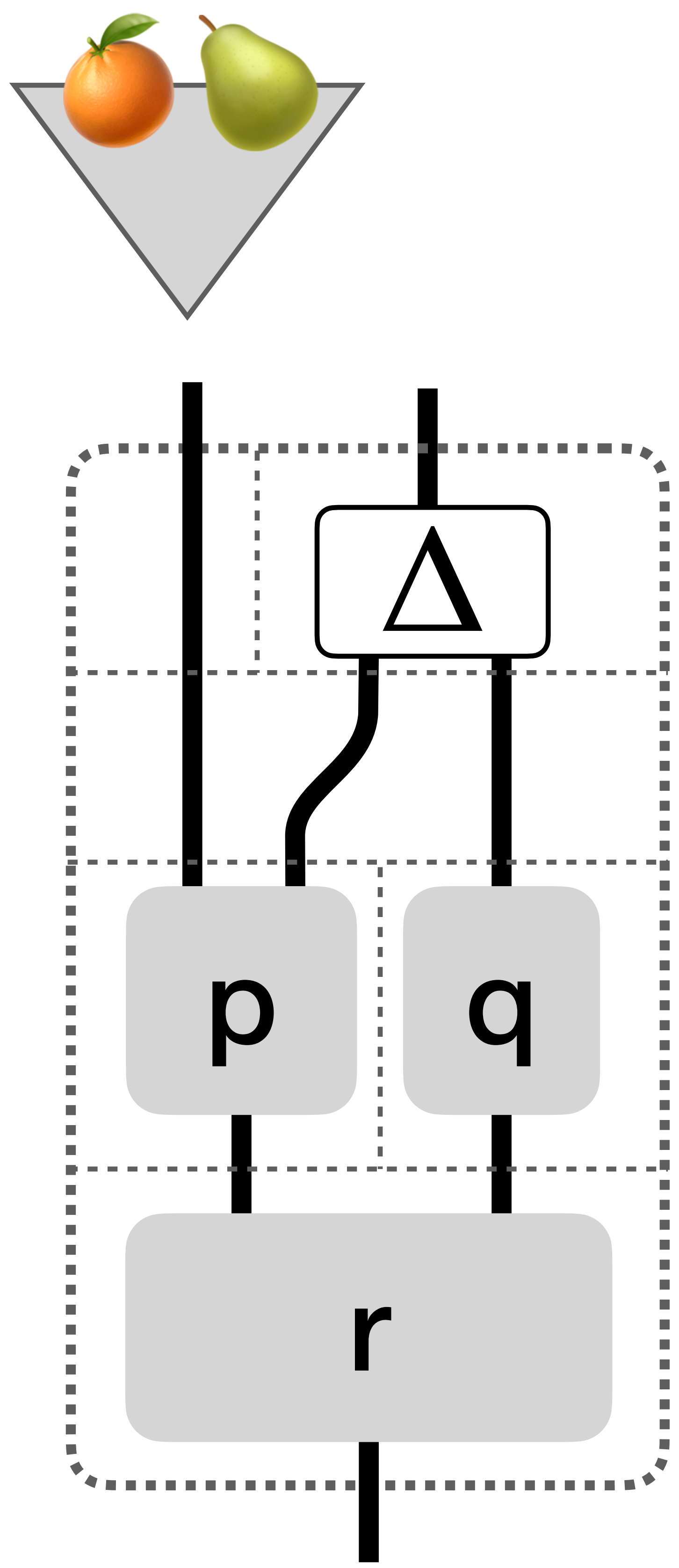

```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
  eliminate(🍅, from = b)
```



Peek into `libretto-lambda`




```scala
// approximately
def delambdify[A, B](
  f: Expr[A] => Expr[B]
): Flow[A, B] | ... =
  val 🍅 : Expr[A] = Var(freshId())
  val b  : Expr[B] = f(🍅)
  eliminate(🍅, from = b)
```



Breaking It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do?
2. What does **`switch`** do?
3. What do the **extractors** do?
   (`ForOffice`, `Monitor`, …)

Breaking It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do?
3. What do the **extractors** do? (`ForOffice`, `Monitor`, …)

Breaking It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do?
3. What do the **extractors** do? 👉
   (`ForOffice`, `Monitor`, ...)

Enum Extractors

```
type Request = Enum
  [ "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]
```

Enum Extractors

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:
```

Enum Extractors

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:

  val ForOffice     : Extractor[Request, Equipment ** DeskLocation]
```

Enum Extractors

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:

  val ForOffice     : Extractor[Request, Equipment ** DeskLocation]
```

Knows the partitioning of a type (`Request`) into disjoint cases.
Represents one partition (`"ForOffice"`).

Enum Extractors

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:

  val ForOffice     : Extractor[Request, Equipment ** DeskLocation]
    = Enum.partition[Request][“ForOffice"]
```

Knows the partitioning of a type (`Request`) into disjoint cases.
Represents one partition (`"ForOffice"`).

Enum Extractors

```
type Request = Enum
  [ "ForOffice"    :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:

  val ForOffice    : Extractor[Request, Equipment ** DeskLocation]
    = Enum.partition[Request][“ForOffice"]

  val WorkFromHome : Extractor[Request, Equipment ** DeliveryAddress]
```

Knows the partitioning of a type (`Request`) into disjoint cases.
Represents one partition (`"ForOffice"`).

Enum Extractors

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:

  val ForOffice     : Extractor[Request, Equipment ** DeskLocation]
    = Enum.partition[Request][“ForOffice"]

  val WorkFromHome : Extractor[Request, Equipment ** DeliveryAddress]
    = Enum.partition[Request]["WorkFromHome"]
```

Knows the partitioning of a type (`Request`) into disjoint cases.
Represents one partition (`"ForOffice"`).

Enum Extractors

```
type Request = Enum
  [ "ForOffice"     :: (Equipment ** DeskLocation)
  || "WorkFromHome" :: (Equipment ** DeliveryAddress)
  ]

object Request:

  val ForOffice     : Extractor[Request, Equipment ** DeskLocation]
    = Enum.partition[Request][“ForOffice"]

  val WorkFromHome : Extractor[Request, Equipment ** DeliveryAddress]
    = Enum.partition[Request]["WorkFromHome"]
```

Knows the partitioning of a type (`Request`) into disjoint cases.
Represents one partition (`"ForOffice"`).

```
import libretto.lambda.EnumModule

val Enum = EnumModule[Flow, **, Enum, ||, ::](using ...)
```

Extractors: What Do They *Do?*

🧑‍💻

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

Extractors: What Do They *Do?*

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

Extractors: What Do They *Do?*

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

```
extension [A, B](ext: Extractor[A, B])
  def unapply(a: Expr¹[A]): Some[Expr¹[B]] =
```

Extractors: What Do They *Do*?

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

```
extension [A, B](ext: Extractor[A, B])
  def unapply(a: Expr¹[A]): Some[Expr¹[B]] =
```

- (at Scala level) always matches

Extractors: What Do They *Do?*

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

```
extension [A, B](ext: Extractor[A, B])
  def unapply(a: Expr¹[A]): Some[Expr¹[B]] =
    val b = Expr¹.Map(a, ext.toFlow¹)
    Some(b)
```

- (at Scala level) always matches

Extractors: What Do They *Do?*

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

```
extension [A, B](ext: Extractor[A, B])
  def unapply(a: Expr¹[A]): Some[Expr¹[B]] =
    val b = Expr¹.Map(a, ext.toFlow¹)
    Some(b)
```

- (at Scala level) always matches
- pretend Extractor is a `Flow¹` (despite being **non-total**)

Extractors: What Do They *Do?*

```scala
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```scala
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

```scala
extension [A, B](ext: Extractor[A, B])
  def unapply(a: Expr¹[A]): Some[Expr¹[B]] =
    val b = Expr¹.Map(a, ext.toFlow¹)
    Some(b)
```

```scala
type Flow¹[A, B] = ...
```

- (at Scala level) always matches
- pretend Extractor is a `Flow¹`
  (despite being **non-total**)
- `Flow¹` = `Flow` + extractors
  allowing illegal (non-total) programs

Extractors: What Do They *Do*?

```
val ForOffice : Extractor[Request, Equipment ** DeskLocation]
```

```
case ForOffice(Monitor(_) ** deskLoc) =>
  requestMonitorFromIT(deskLoc)
```

```
extension [A, B](ext: Extractor[A, B])
  def unapply(a: Expr¹[A]): Some[Expr¹[B]] =
    val b = Expr¹.Map(a, ext.toFlow¹)
    Some(b)
```

```
type Flow¹[A, B] = ...
val lambdas¹: Lambdas[Flow¹, **, …]
type Expr¹[A] = lambdas¹.Expr[A]
```

- (at Scala level) always matches
- pretend Extractor is a `Flow¹` (despite being **non-total**)
- `Flow¹` = `Flow` + extractors allowing illegal (non-total) programs

Breaking It Down

```
Flow { req =>                                     🧑‍💻
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do?
3. What do the **extractors** do?
   (`ForOffice`, `Monitor`, ...)

Breaking It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do?
3. What do the **extractors** do? ✔
   (`ForOffice`, `Monitor`, …)

Breaking It Down

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do? 👈
3. What do the **extractors** do? ✔
   (`ForOffice`, `Monitor`, ...)

switch

```
req switch {
  case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)
  case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc)
  case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)
}
```

switch

👨‍💻

```
req switch {
  case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)
  case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc)
  case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)
}
```

macro-expand(*) to

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

switch

```
req switch {
  case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)
  case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc)
  case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)
}
```

macro-expand(*) to

taking each case as it’s own function

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

switch

```
req switch {
  case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)
  case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc)
  case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)
}
```

macro-expand(*) to

taking each case as it's own function

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

records source position (for error reporting)

switch

```
req switch {
  case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)
  case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc)
  case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)
}
```

macro-expand(*) to

taking each case as it's own function

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

records source position (for error reporting)

(*) not implemented for this demo

`switch`: What Does It *Do*?

👨‍💻

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

```
extension [A](a: Expr[A])
  def switch[R](cases: (Expr[A] => Expr[R])*): Expr[R] =
```

`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

Calls the library 😎

```
extension [A](a: Expr[A])
  def switch[R](cases: (Expr[A] => Expr[R])*): Expr[R] =
    patmat.delambdifyAndCompile(a, cases)
```

switch: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

Calls the library 😎

```
extension [A](a: Expr[A])
  def switch[R](cases: (Expr[A] => Expr[R])*): Expr[R] =
    patmat.delambdifyAndCompile(a, cases)
```

```
import libretto.lambda.PatternMatching
val patmat = PatternMatching[Flow, **].forLambdas(lambdas¹)(...)
```

`switch`: What Does It *Do*?

👨‍💻

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

`switch`: What Does It *Do*?

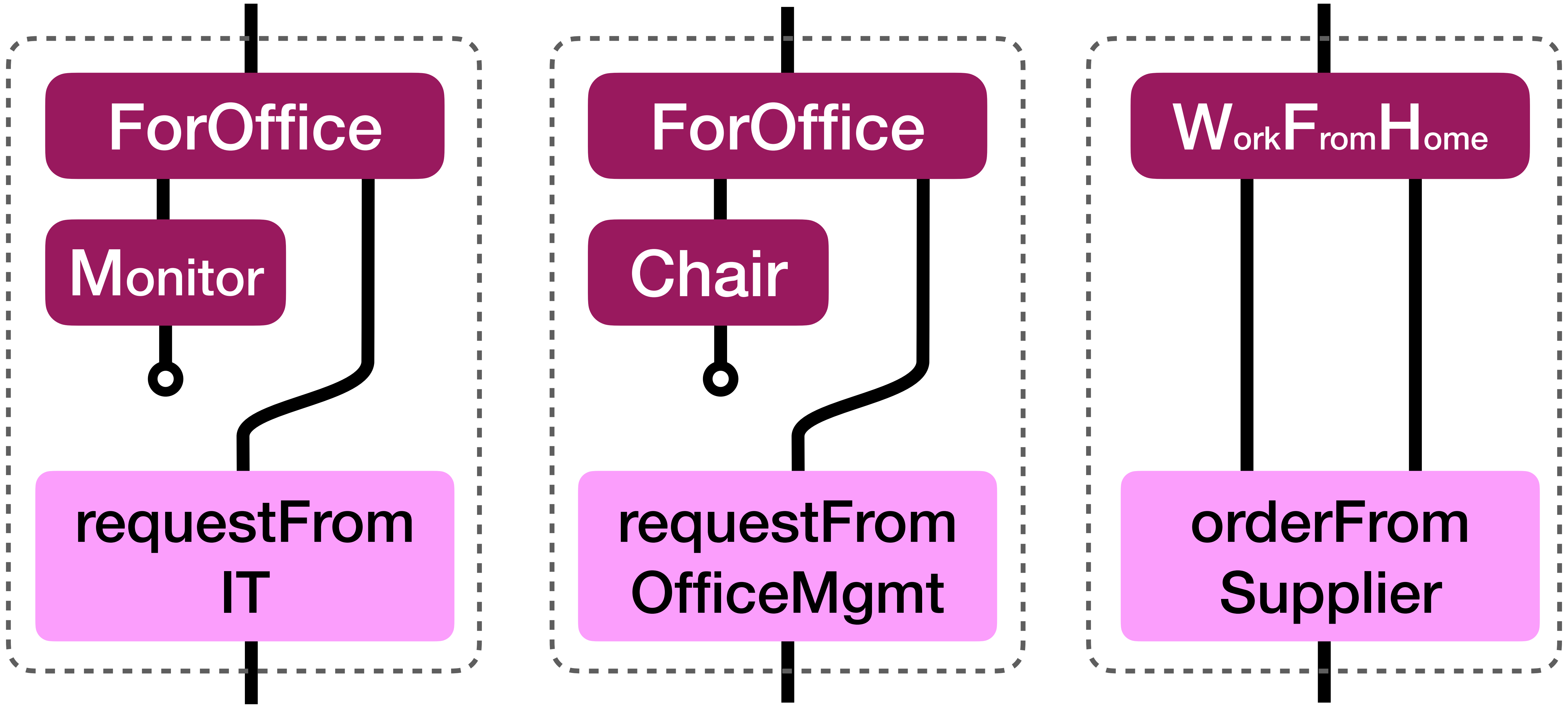
```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case

switch: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

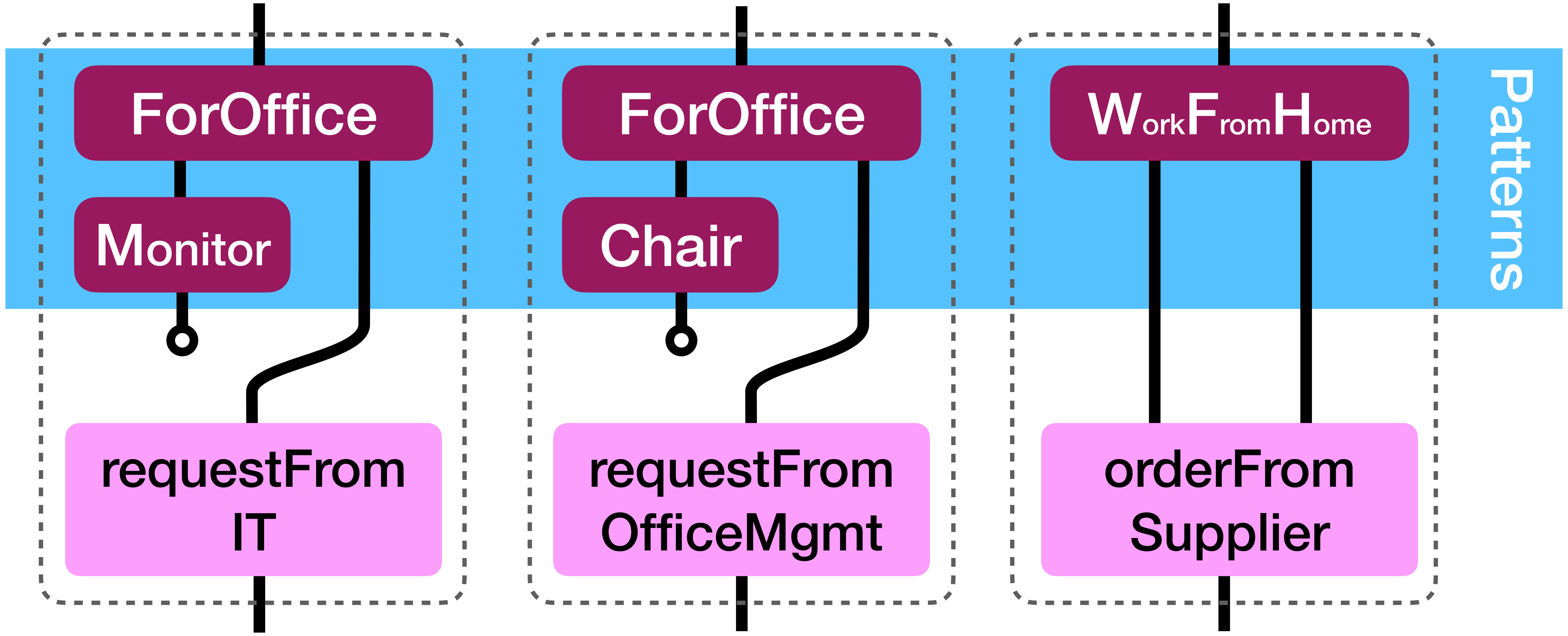
1. Delambdify each case



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case



`switch`: What Does It *Do*?

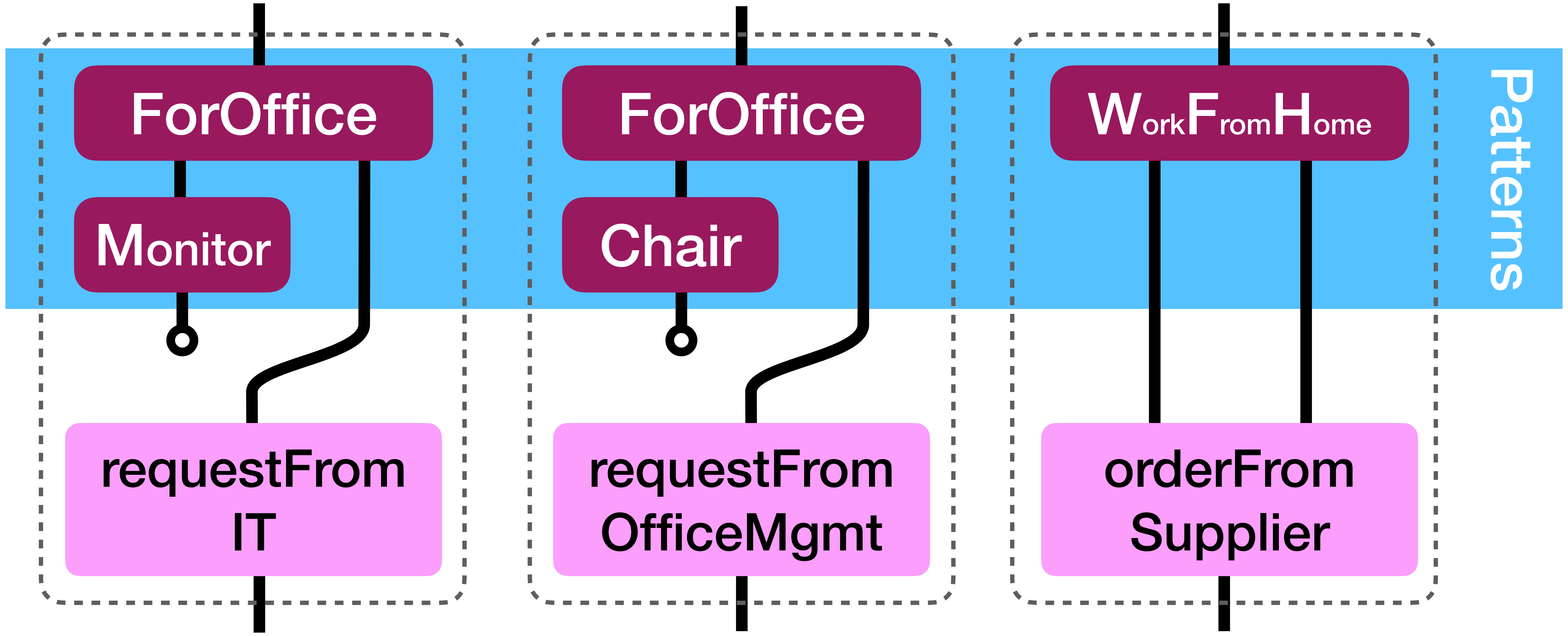
```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)      },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case

Idea: Recombine to

- eliminate (non-total) Extractors
- form (total) Handlers.

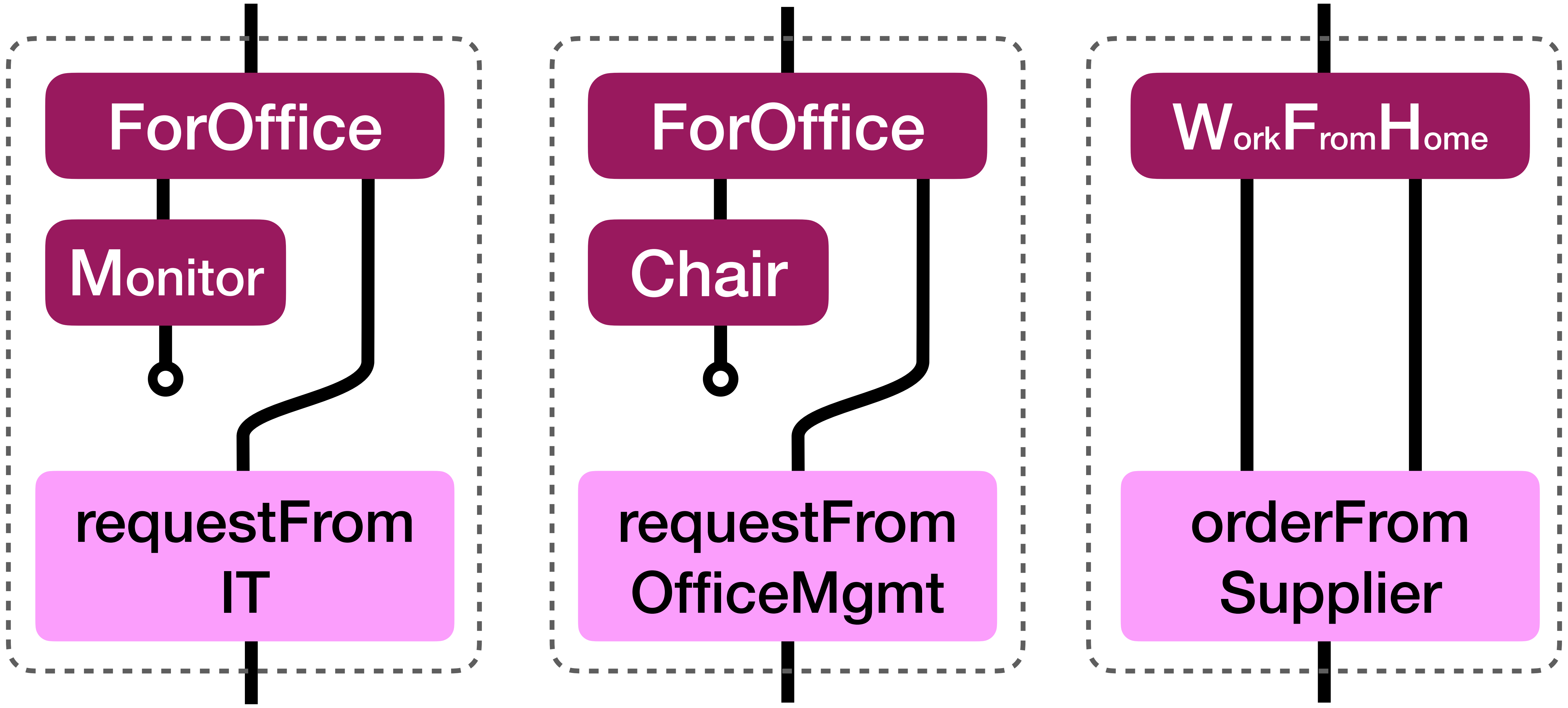
Fail if not possible.



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

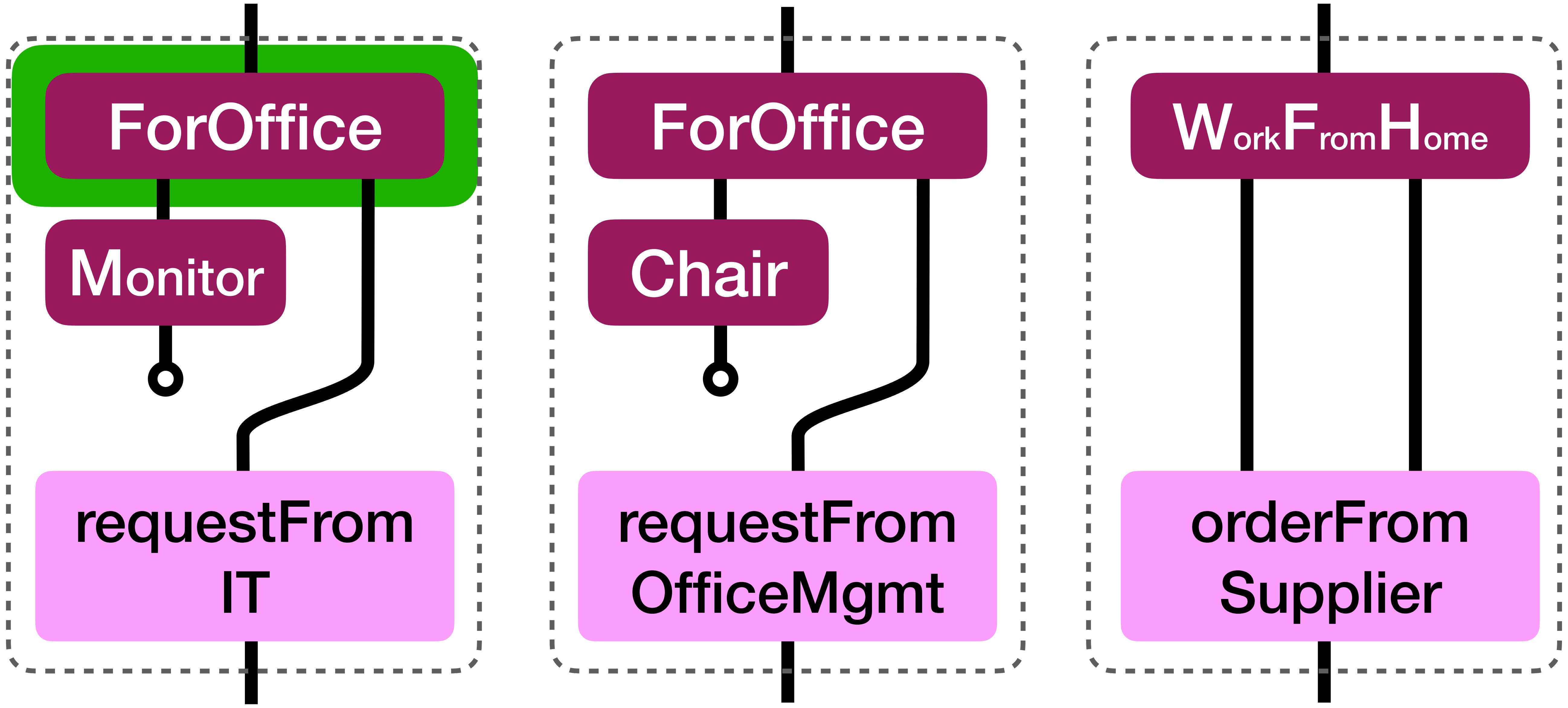
1. Delambdify each case



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

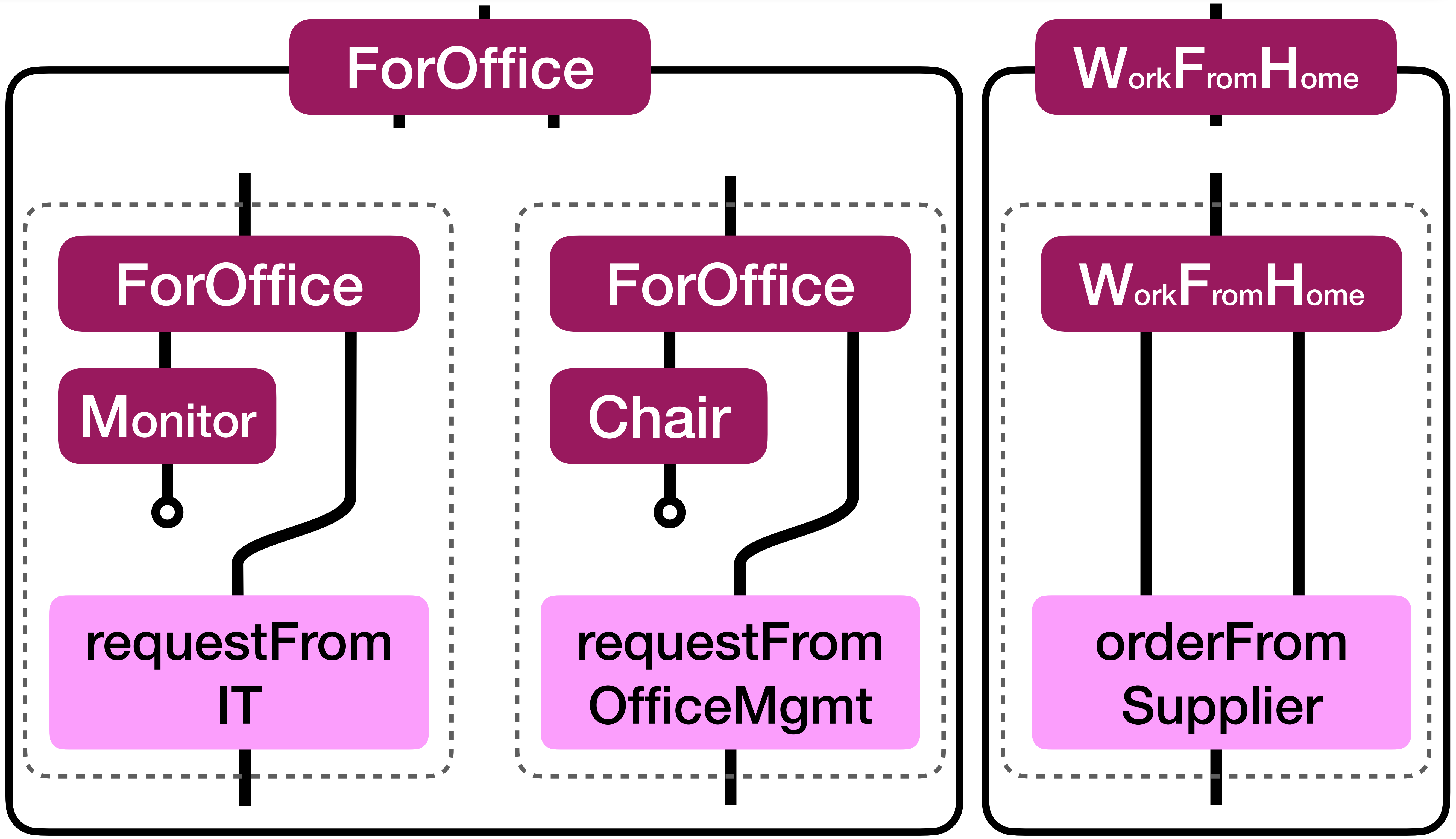
1. Delambdify each case
2. Pick the first Extractor, obtain the whole partitioning



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

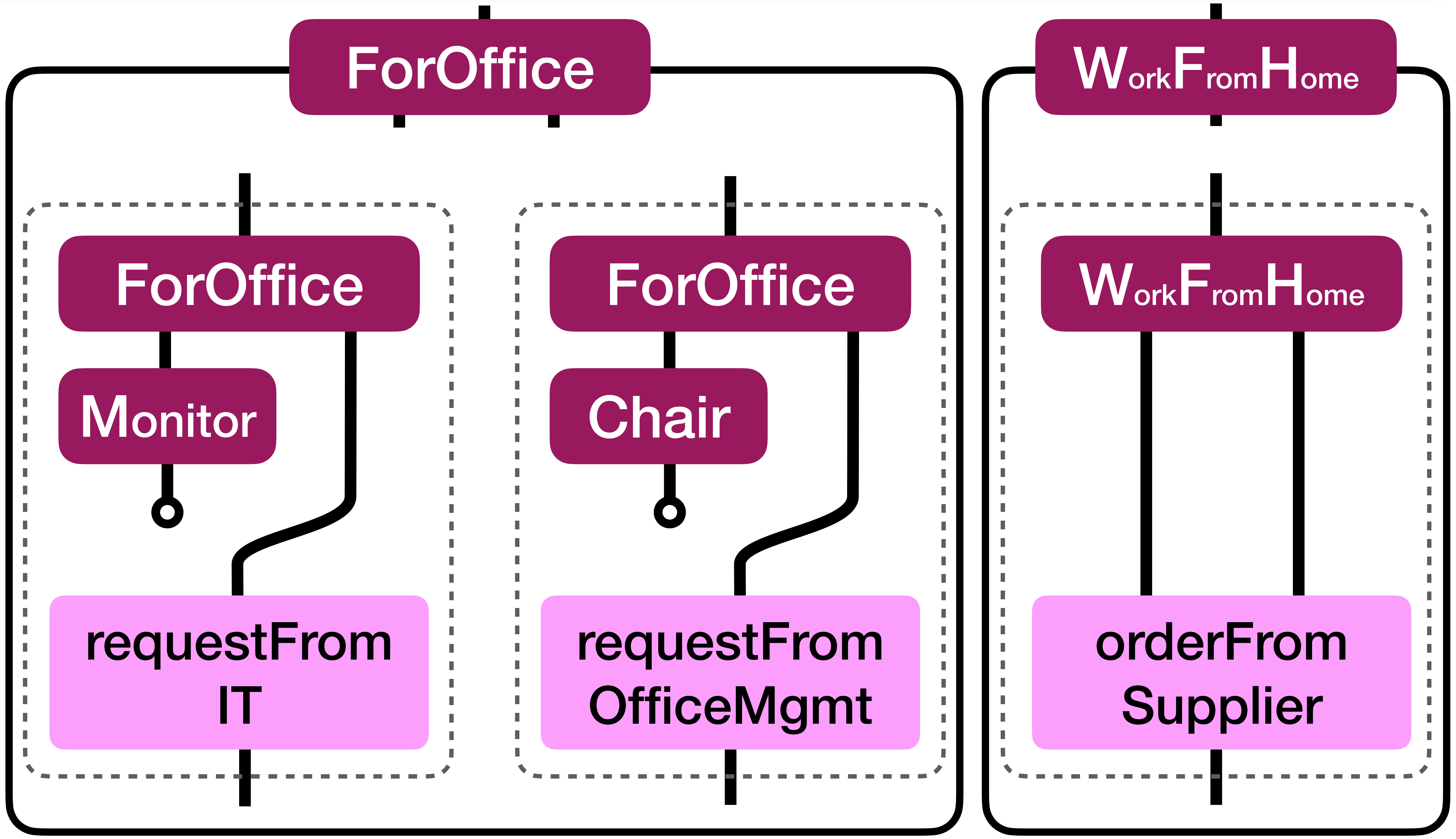
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

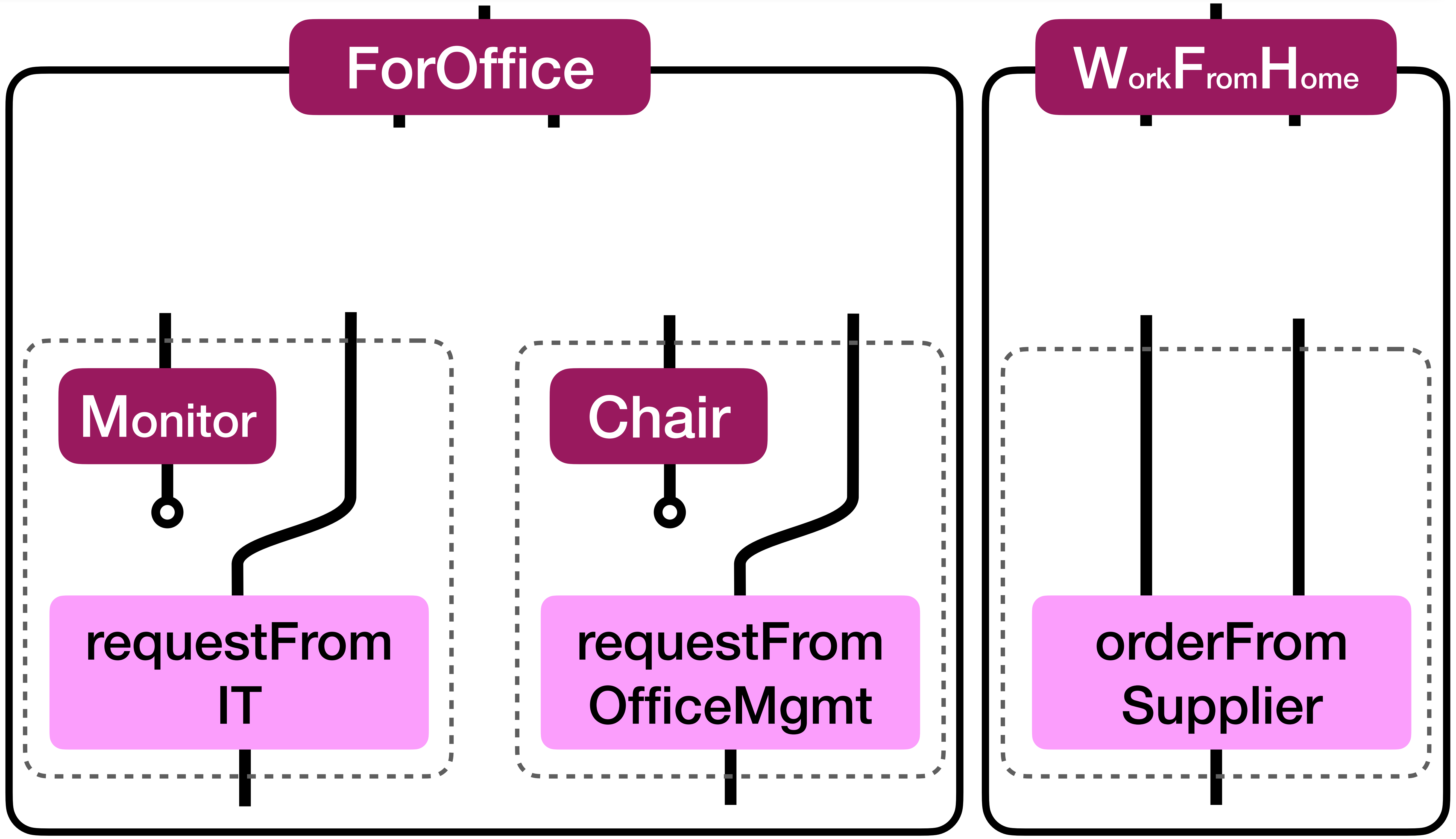
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor



`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

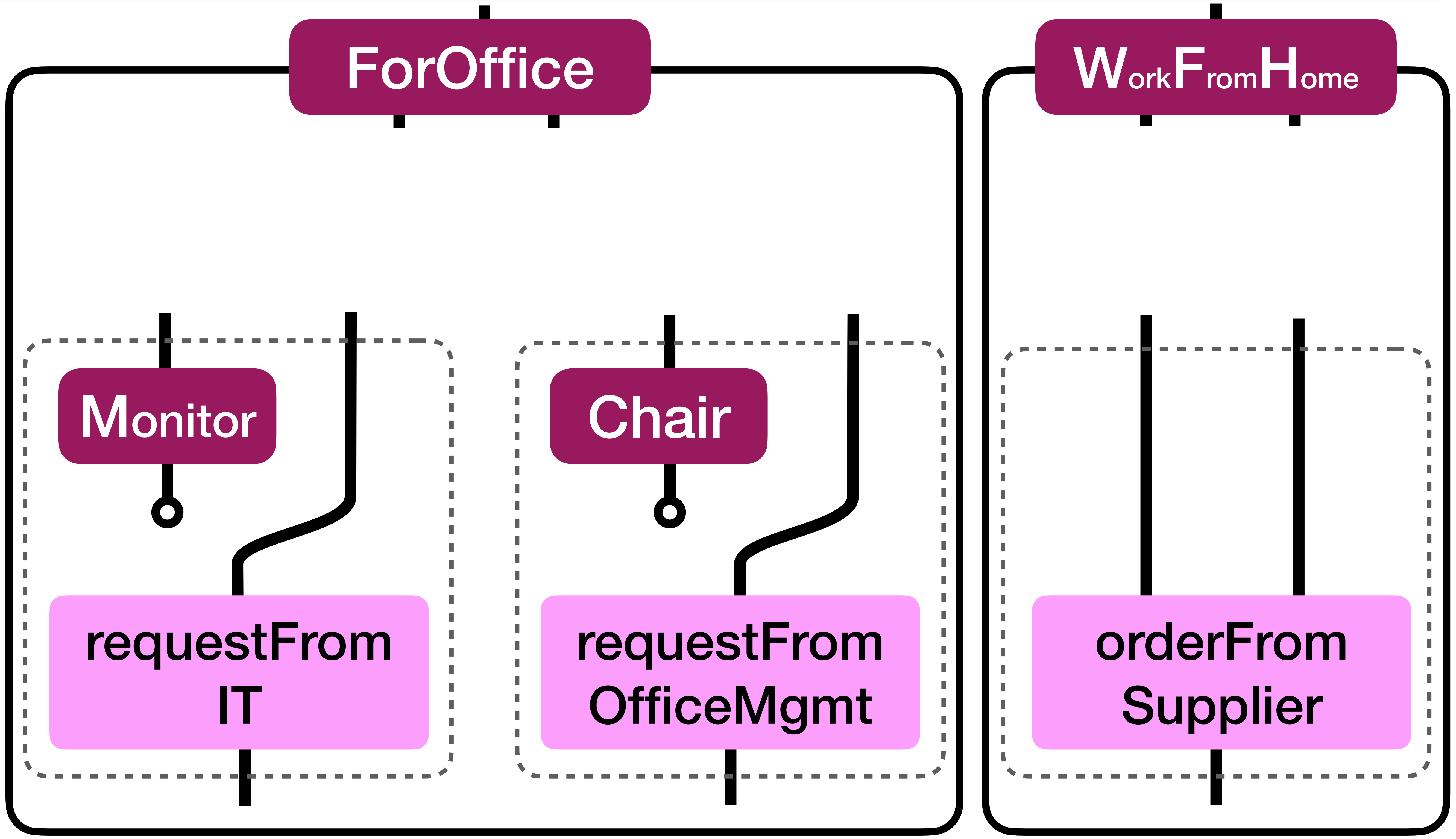
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

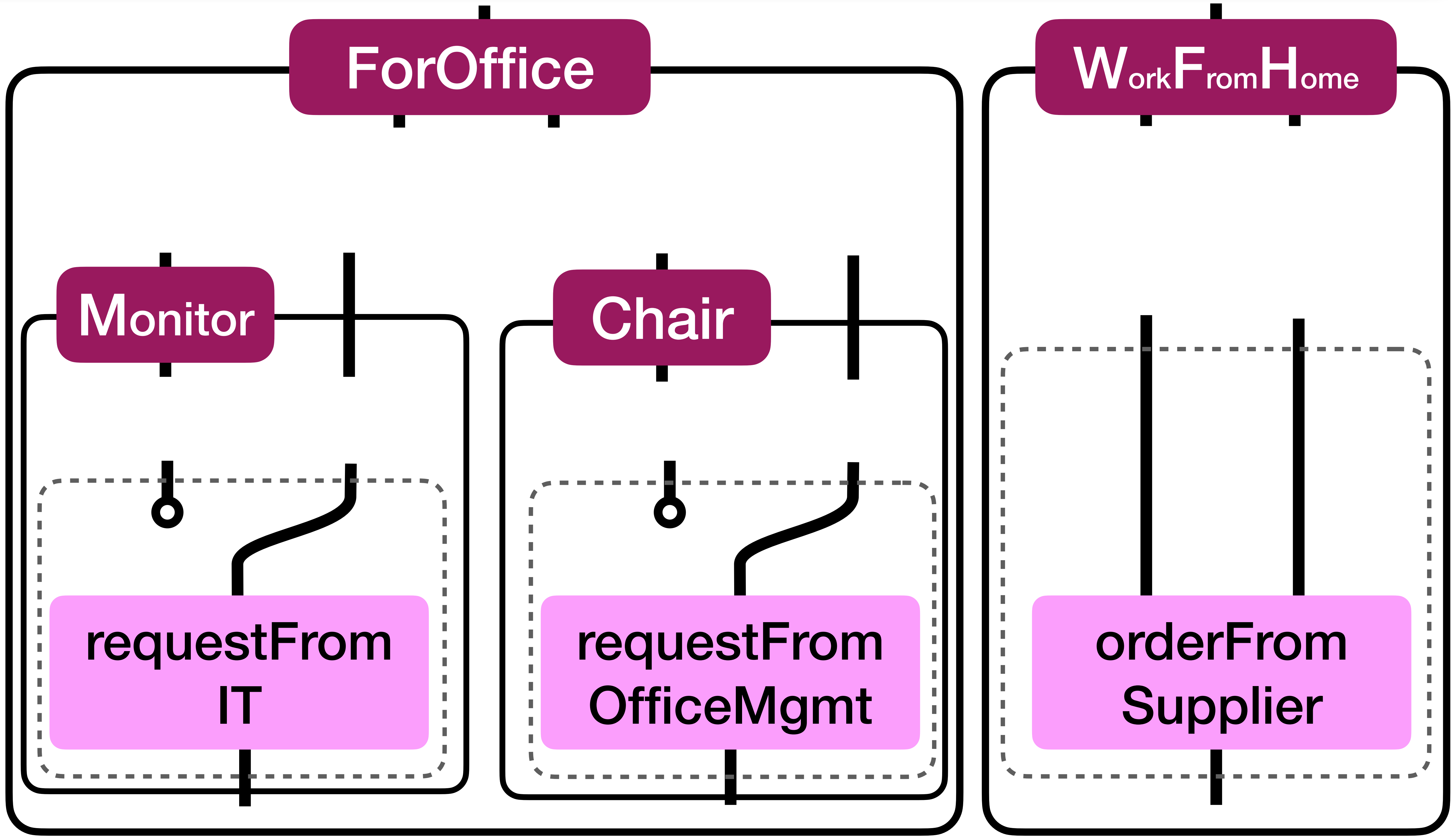
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group



`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

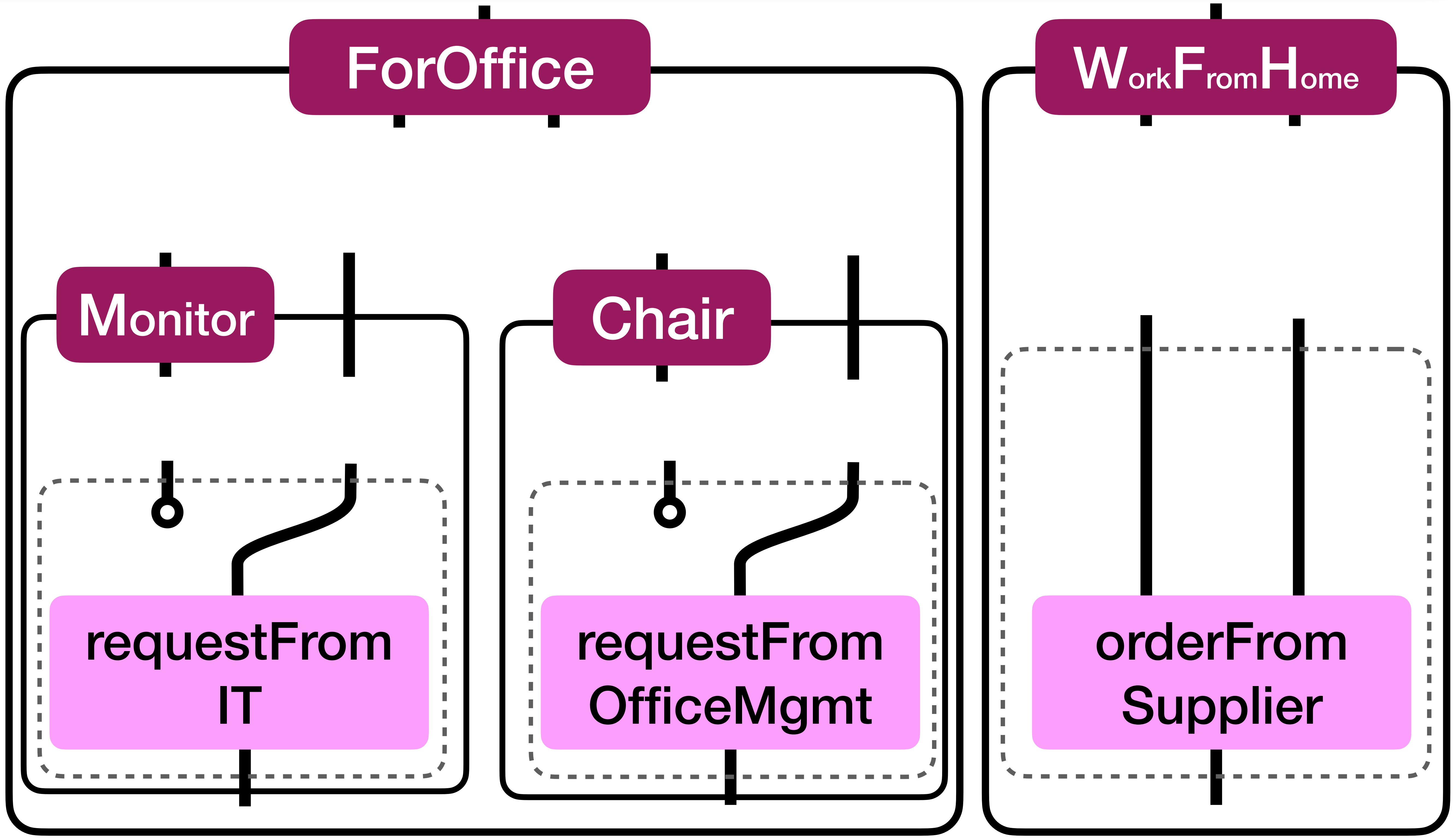
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed

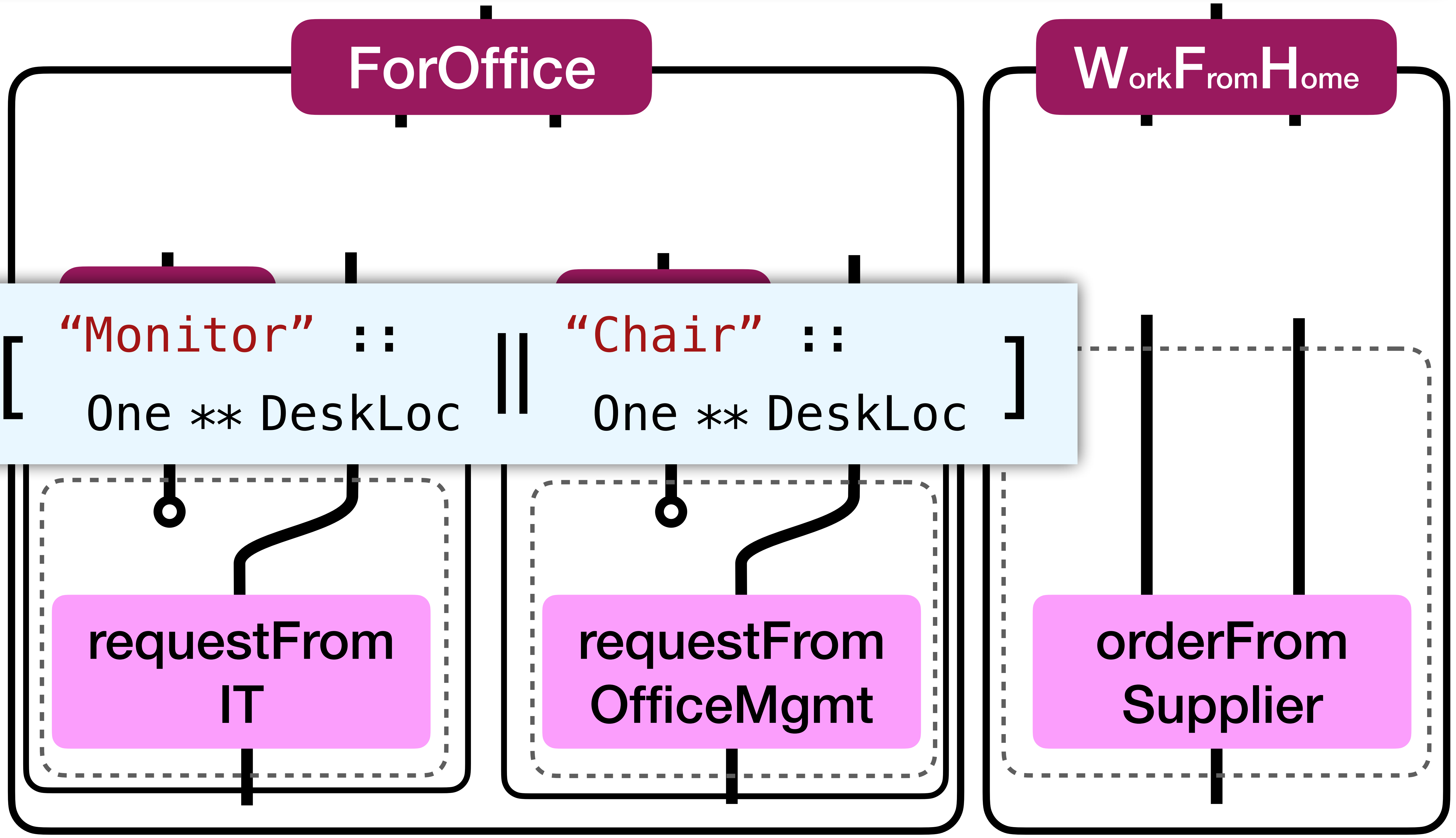


`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaus
4. Remove the matched ex
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed

```
Enum [ "Monitor" :: One ** DeskLoc || "Chair" :: One ** DeskLoc ]
```

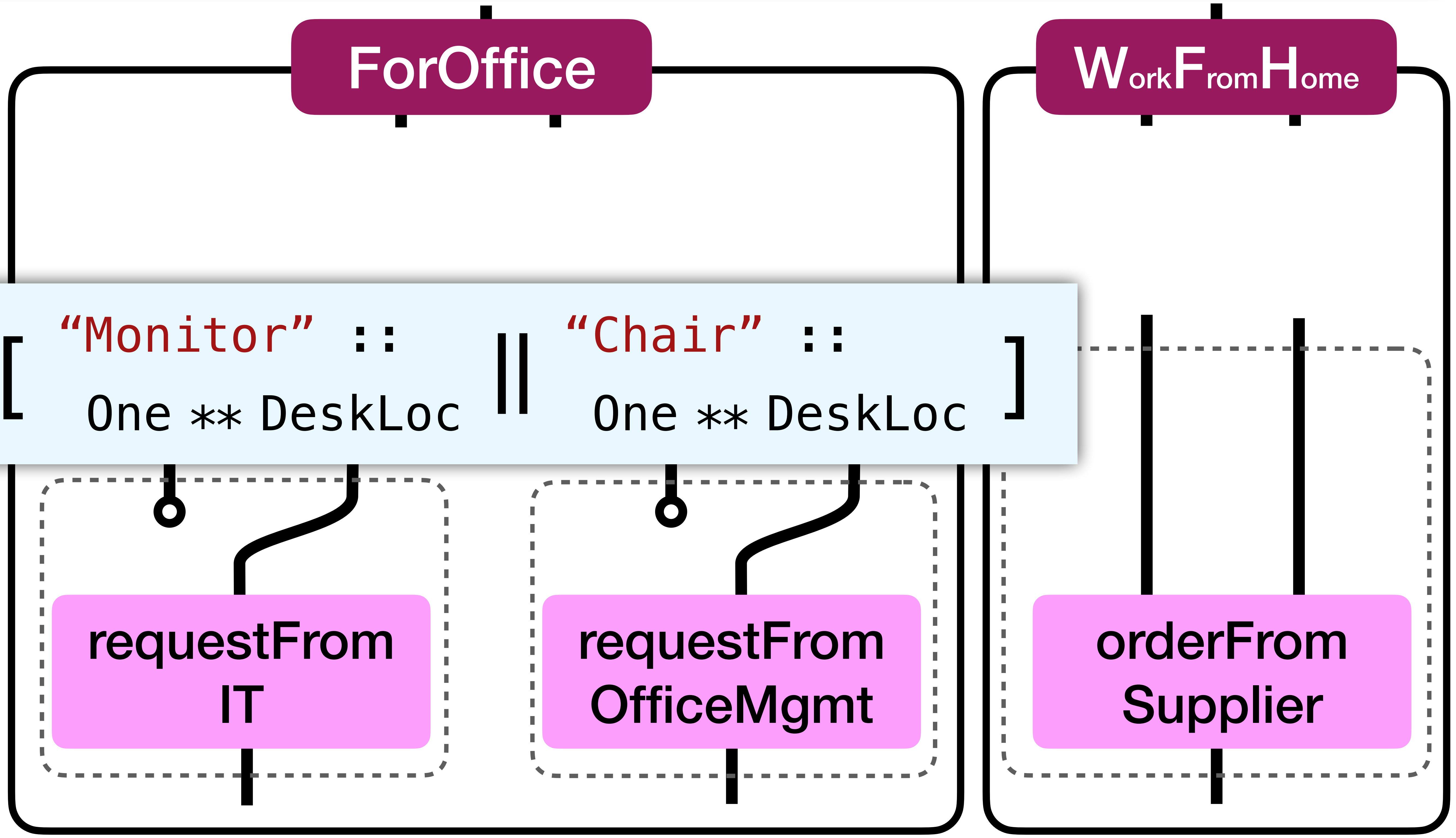


`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhau
4. Remove the matched ex
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed

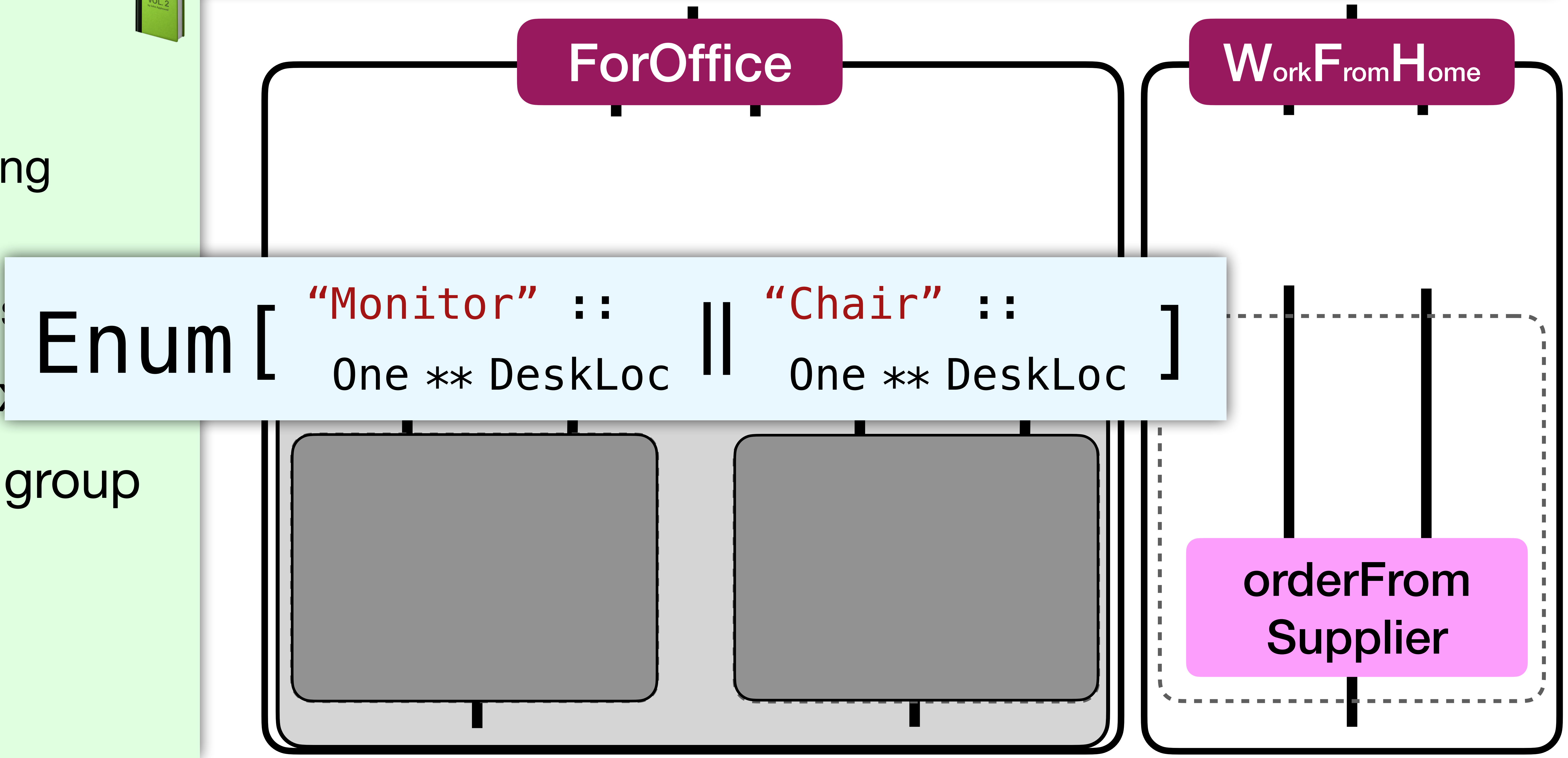
Enum ["Monitor" :: One ** DeskLoc || "Chair" :: One ** DeskLoc]



`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)     => orderFromSupplier(item ** address)  }
)
```

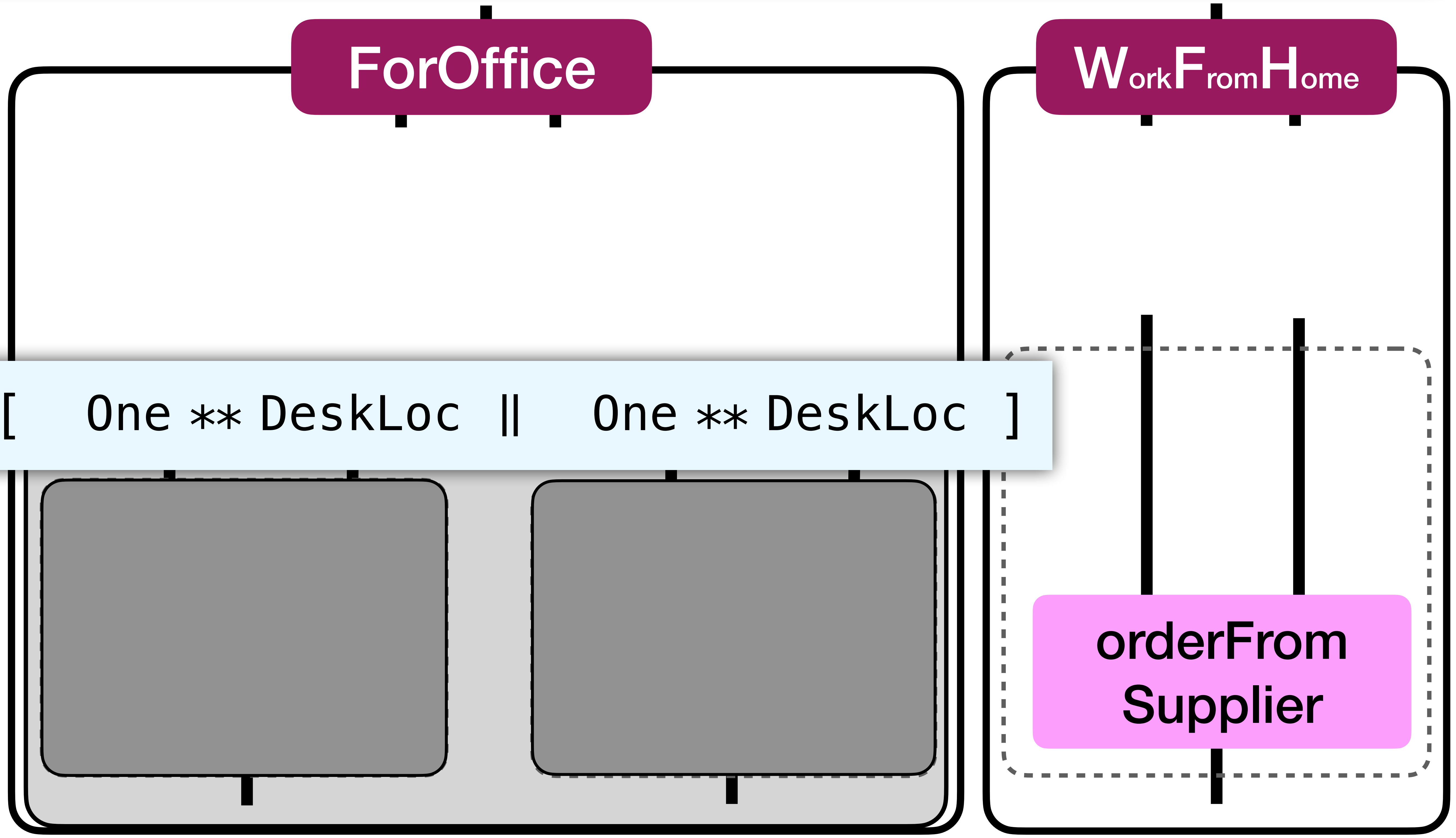
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhau
4. Remove the matched e
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

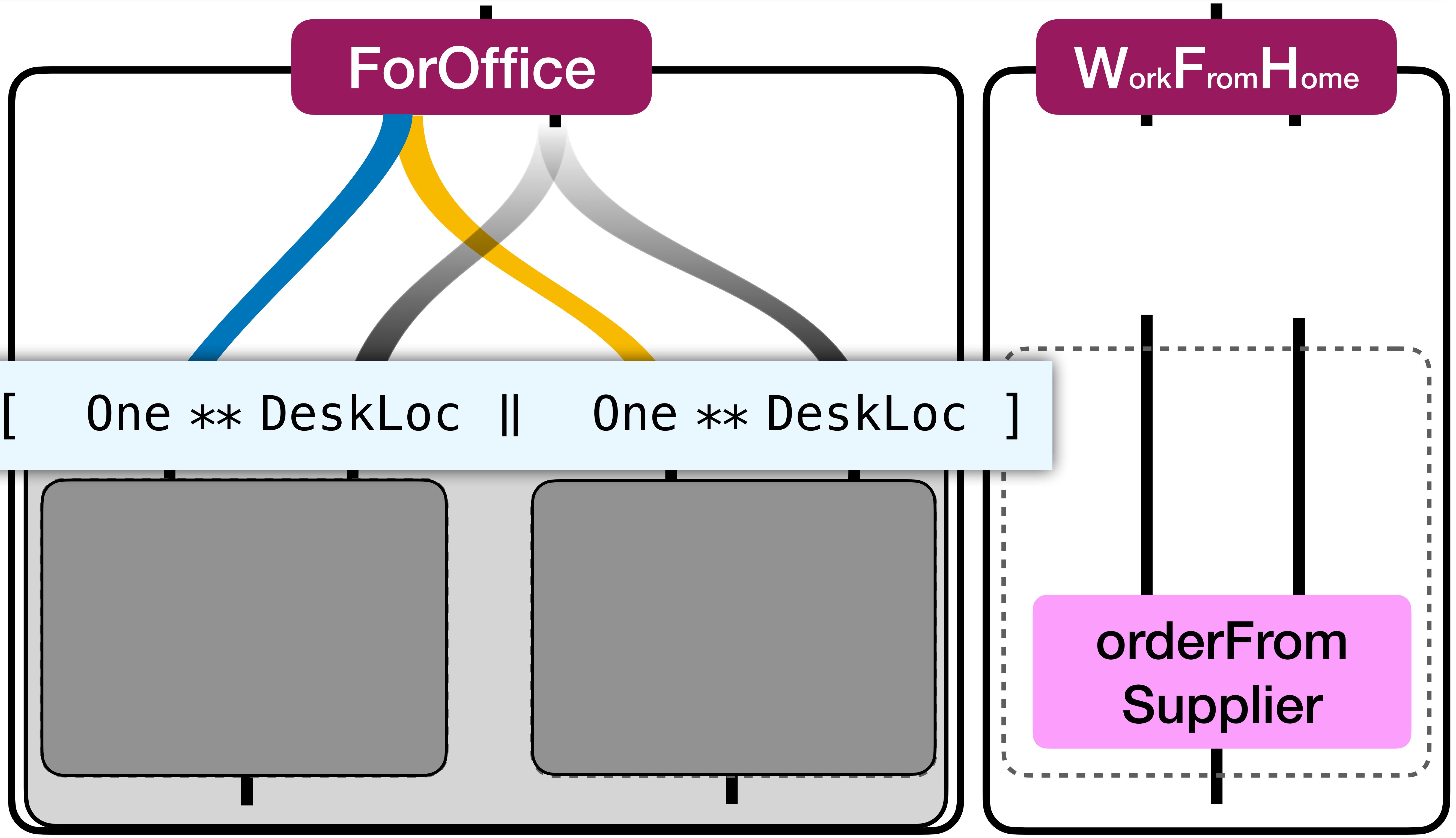
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extrac
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

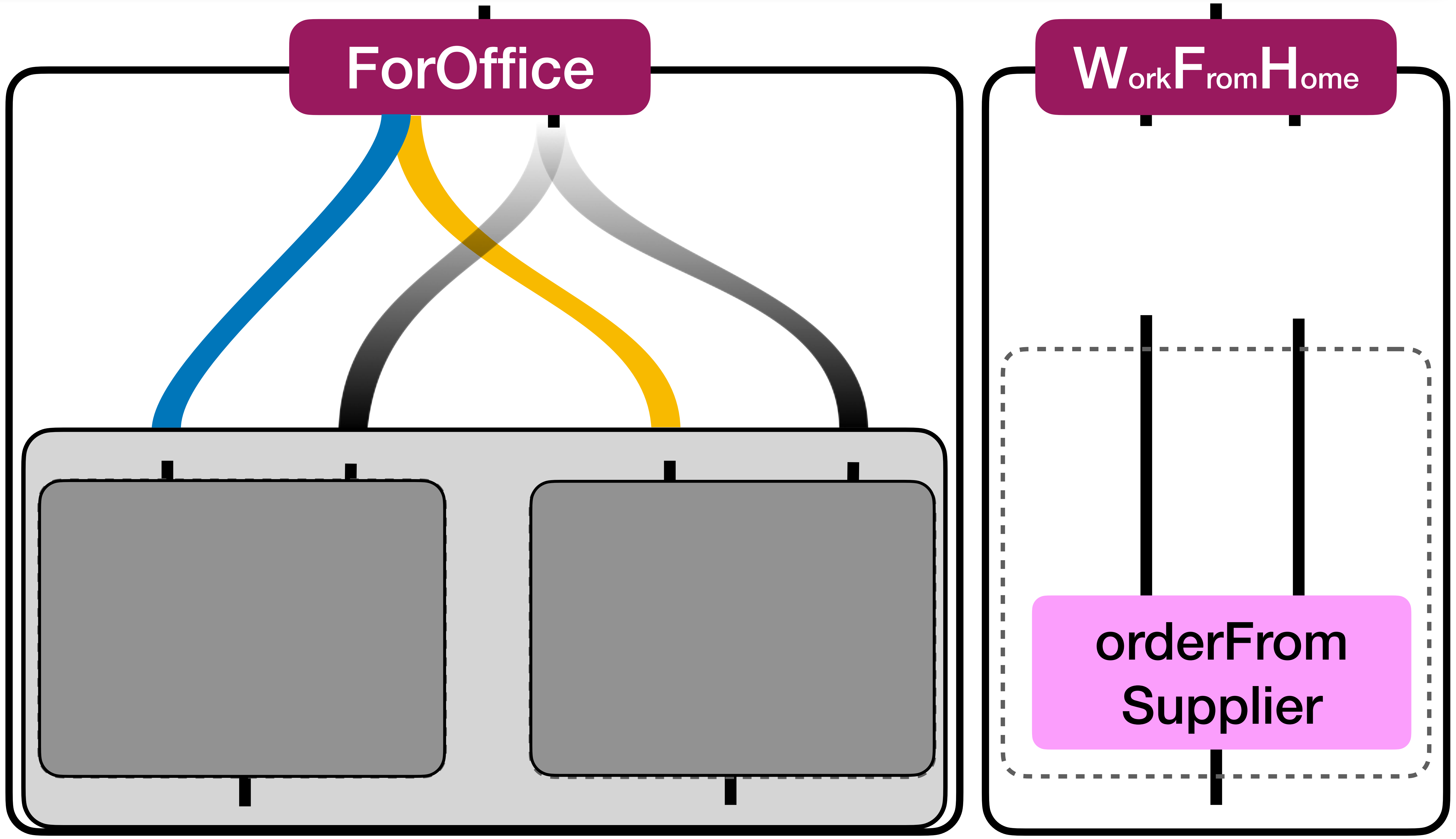
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extrac
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

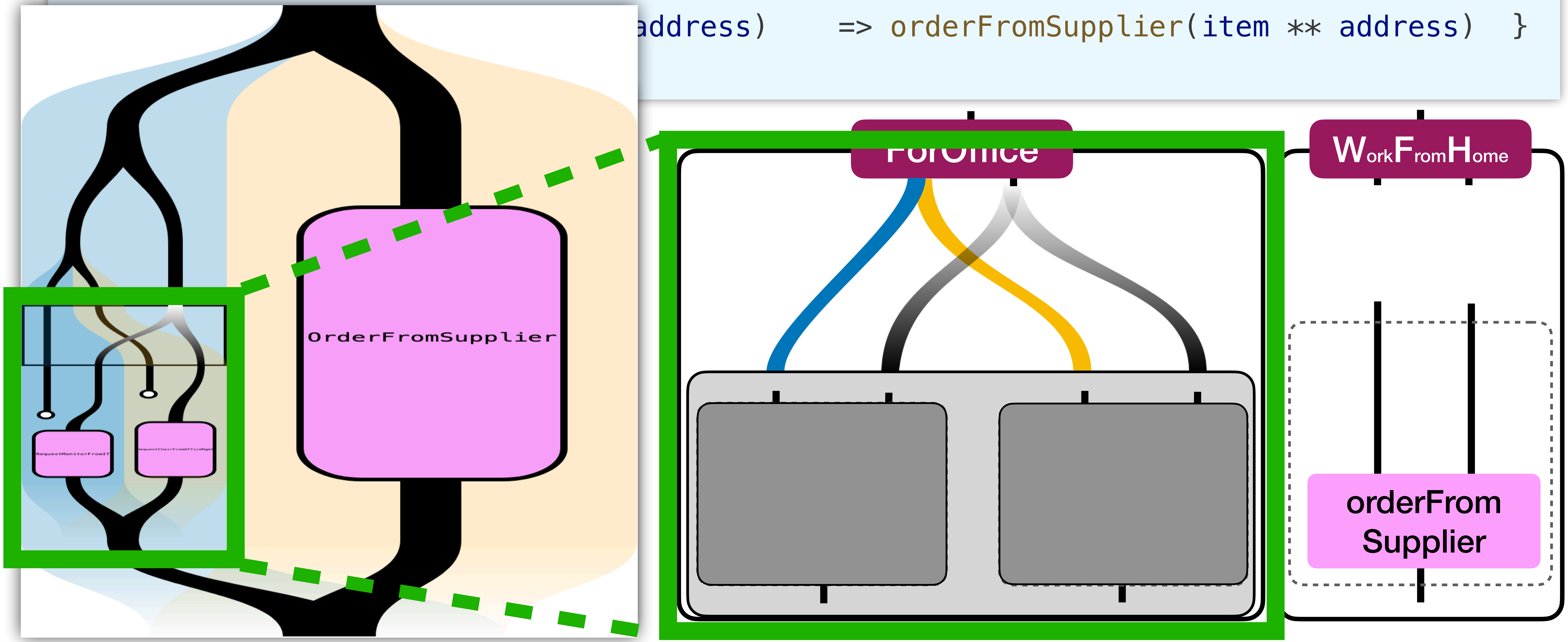
```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

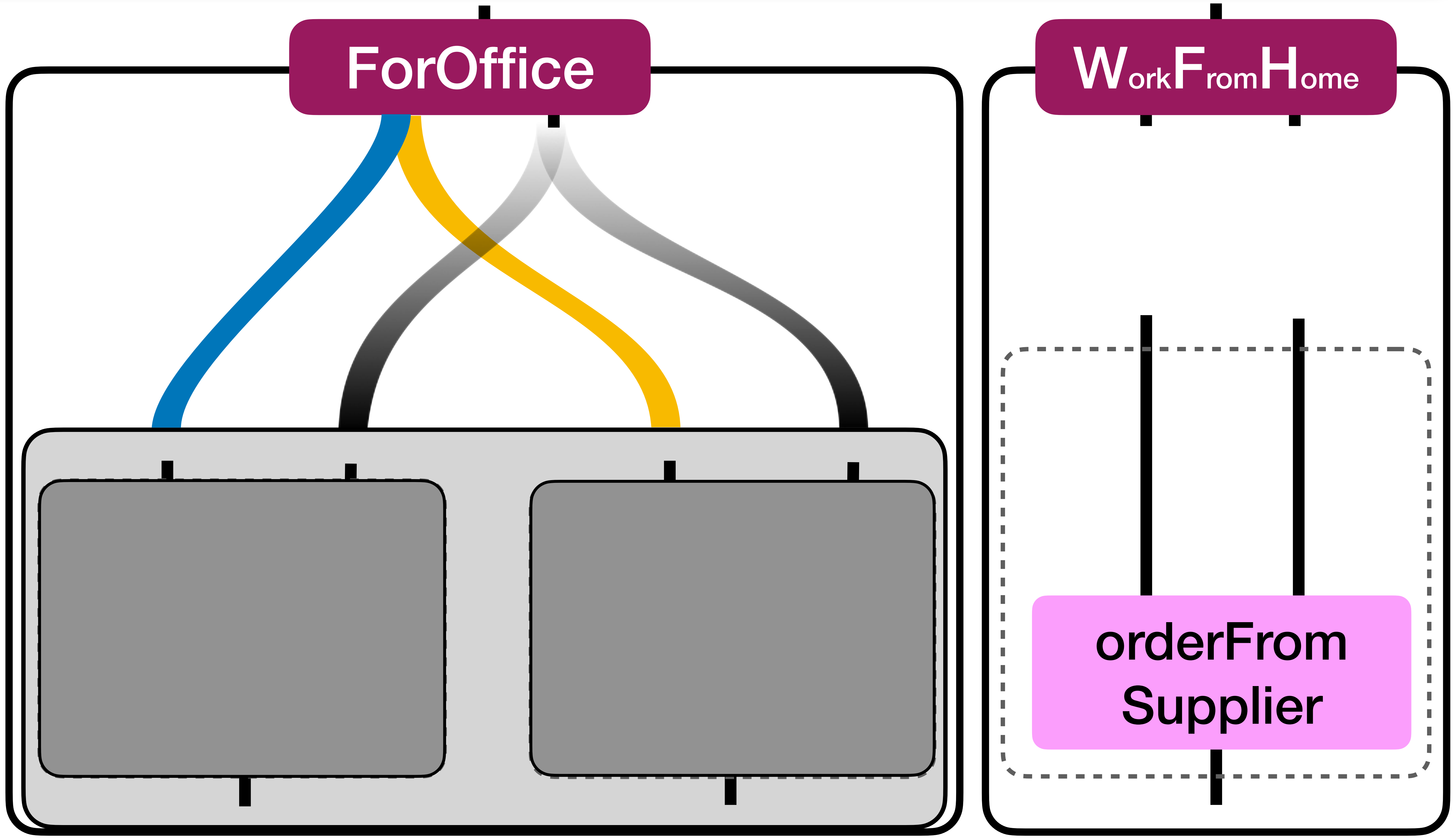
```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
                                     address)    => orderFromSupplier(item ** address)  }
```



`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

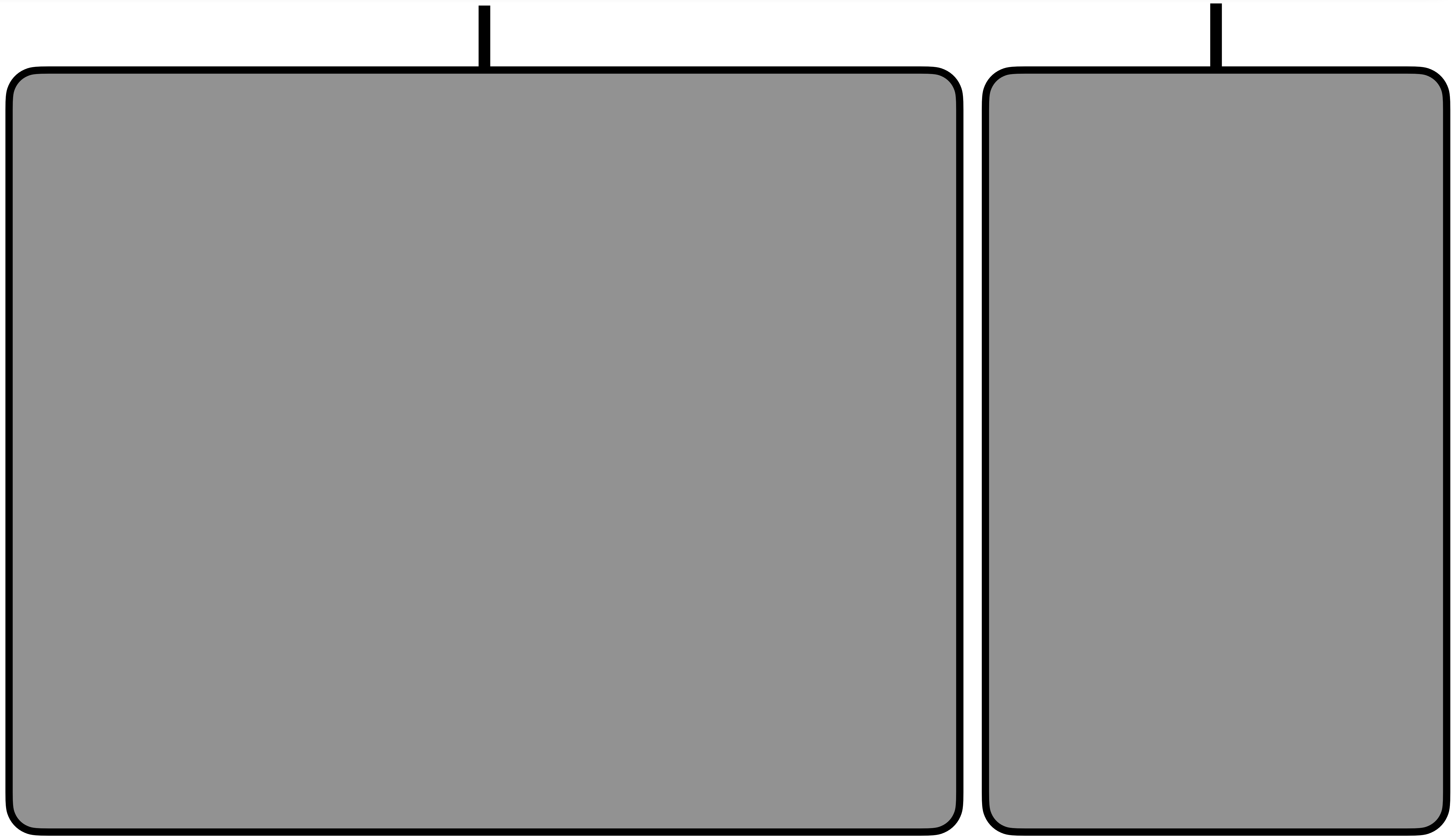
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)       },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

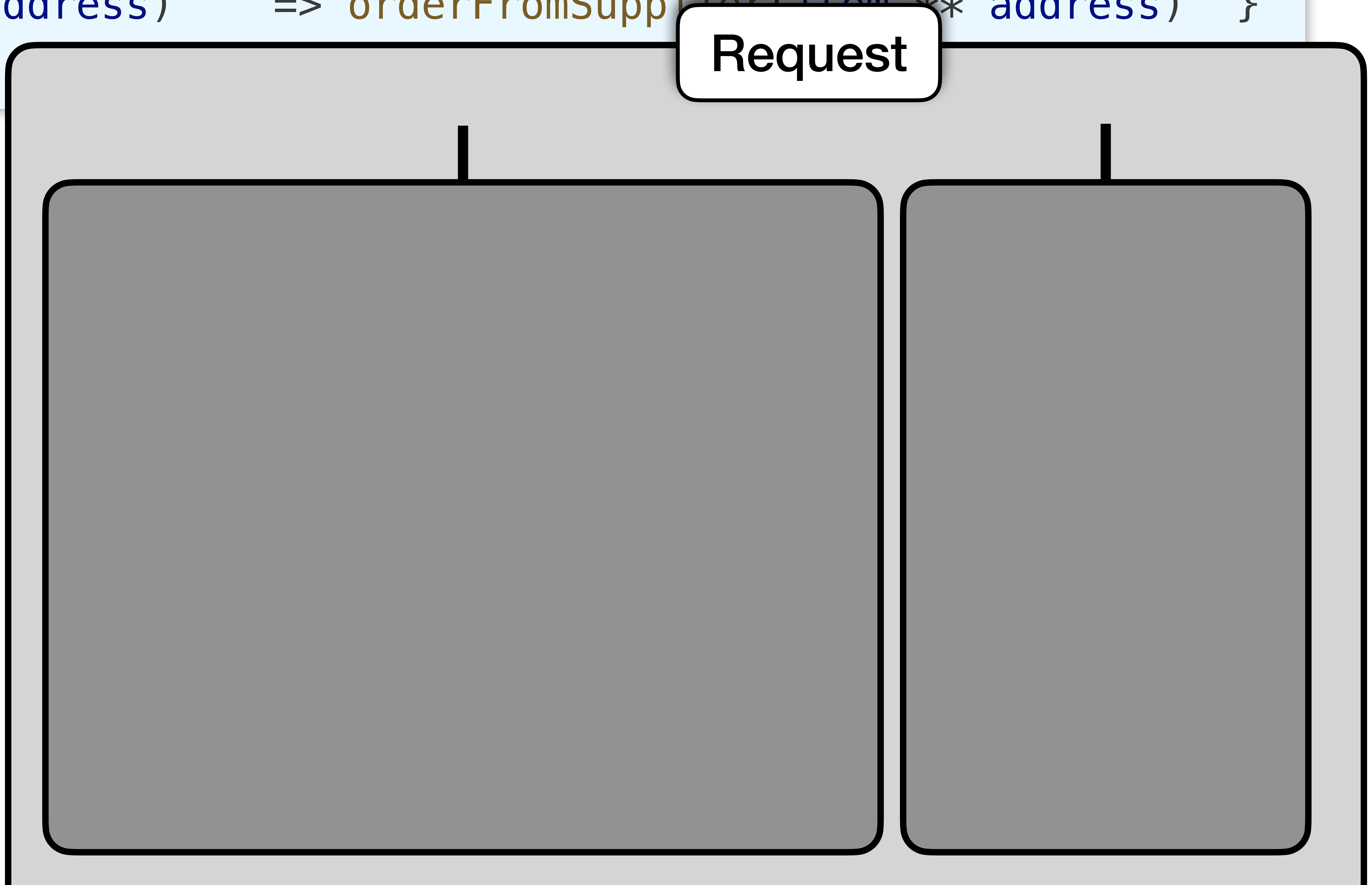
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

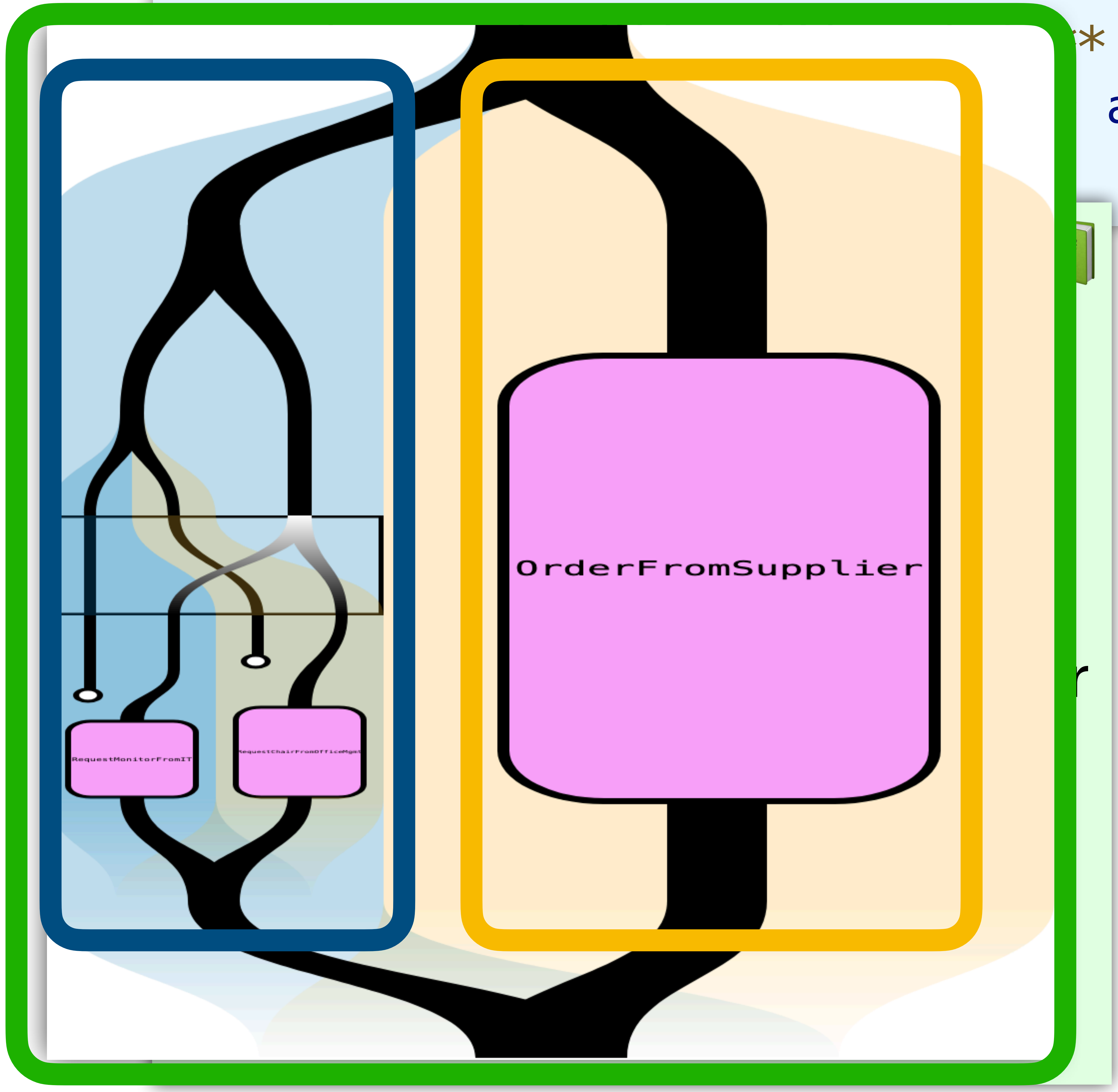
```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

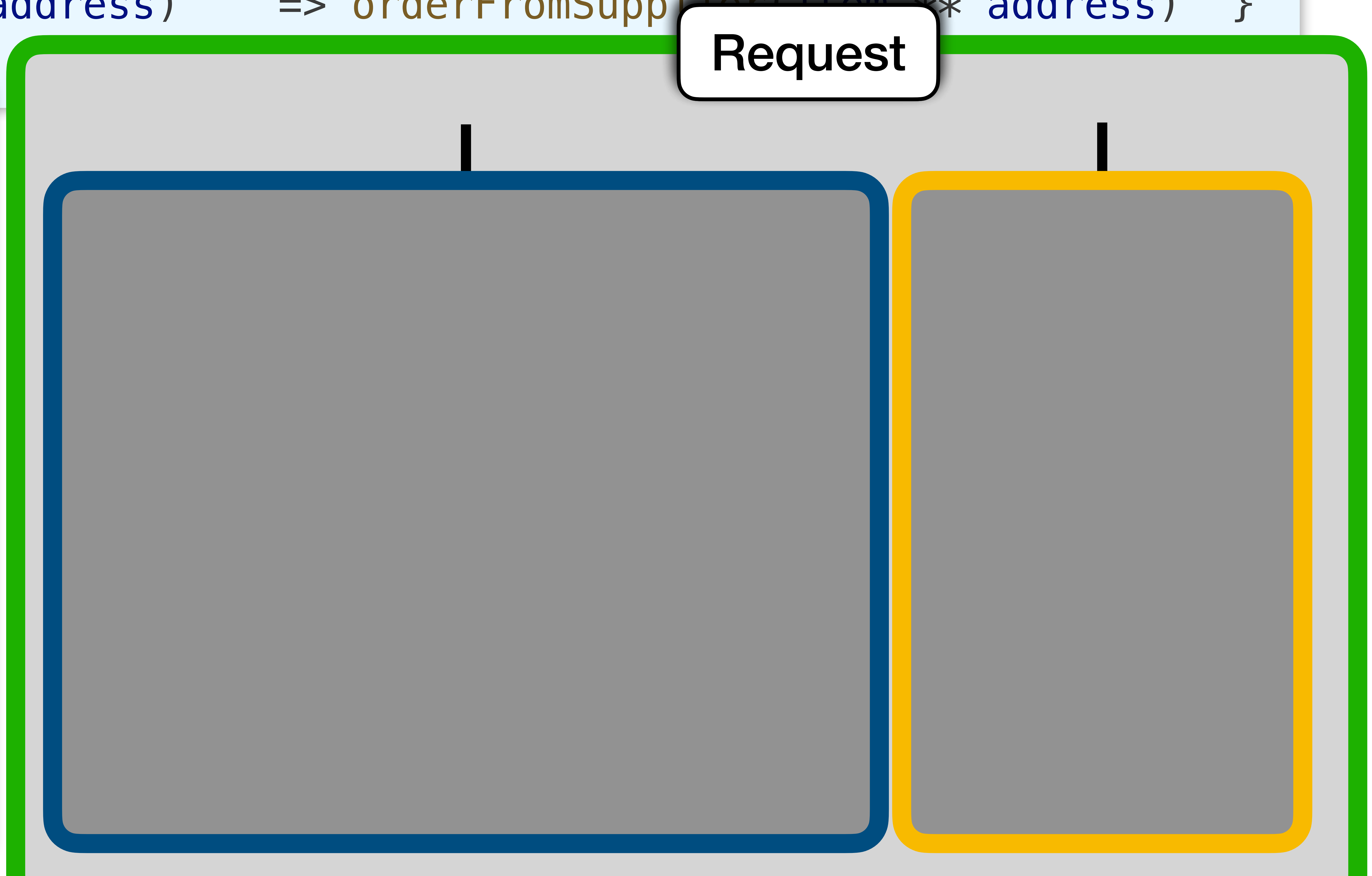
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
                                * deskLoc)    => requestChairFromOfficeMgmt(deskLoc) },
                                address)      => orderFromSuppl                address)  }
```

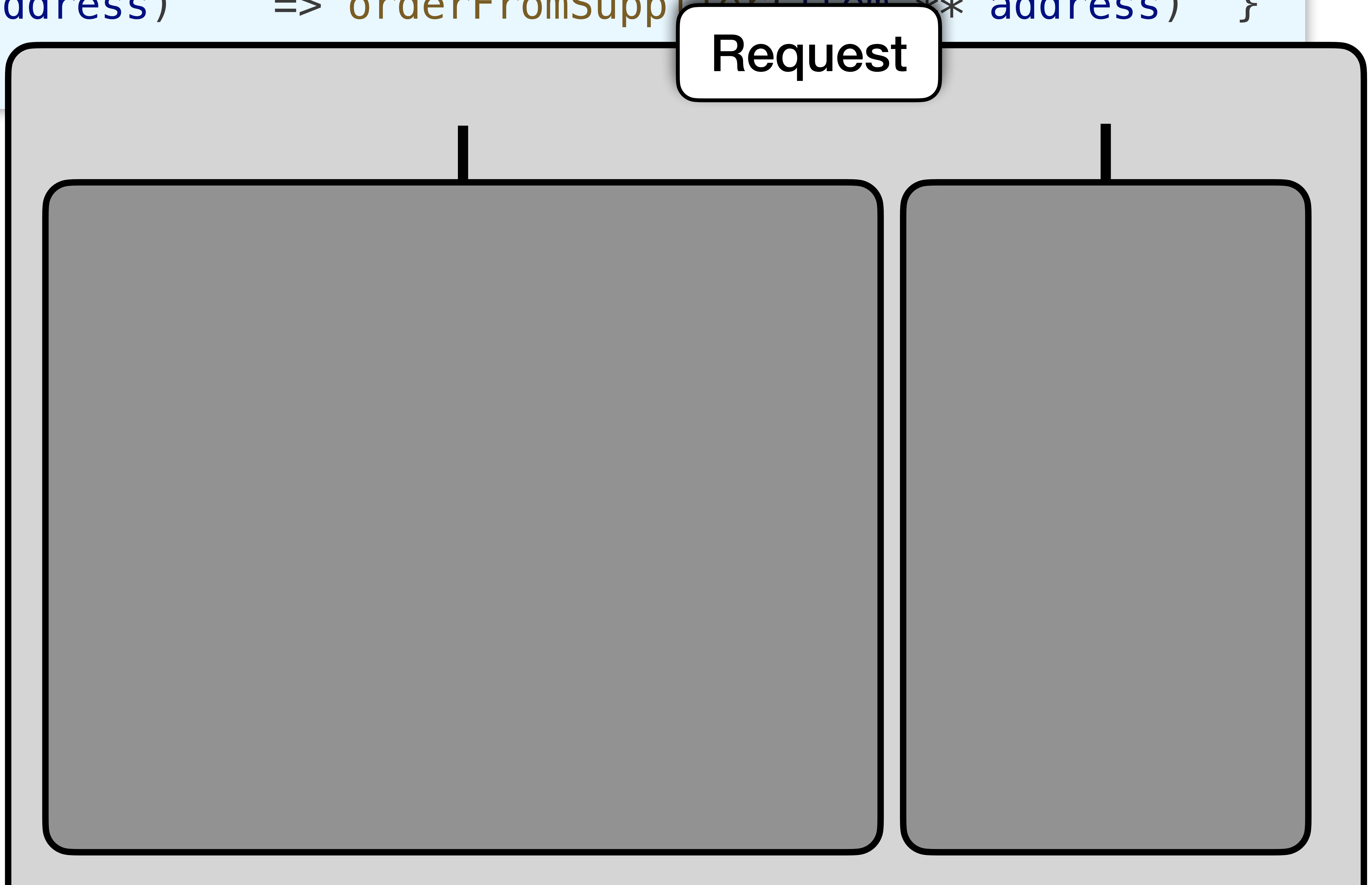




`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)         },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

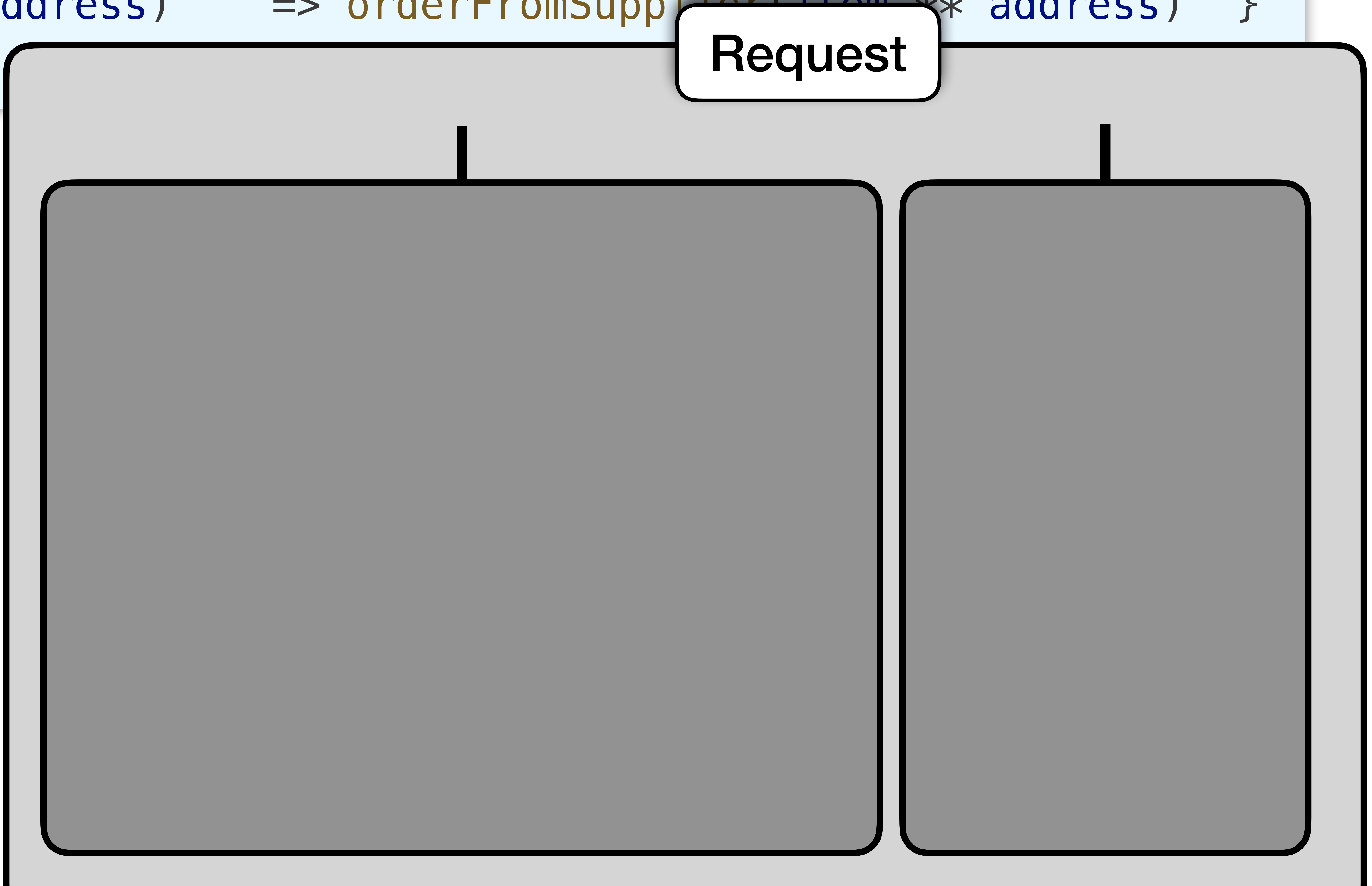
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed



`switch`: What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

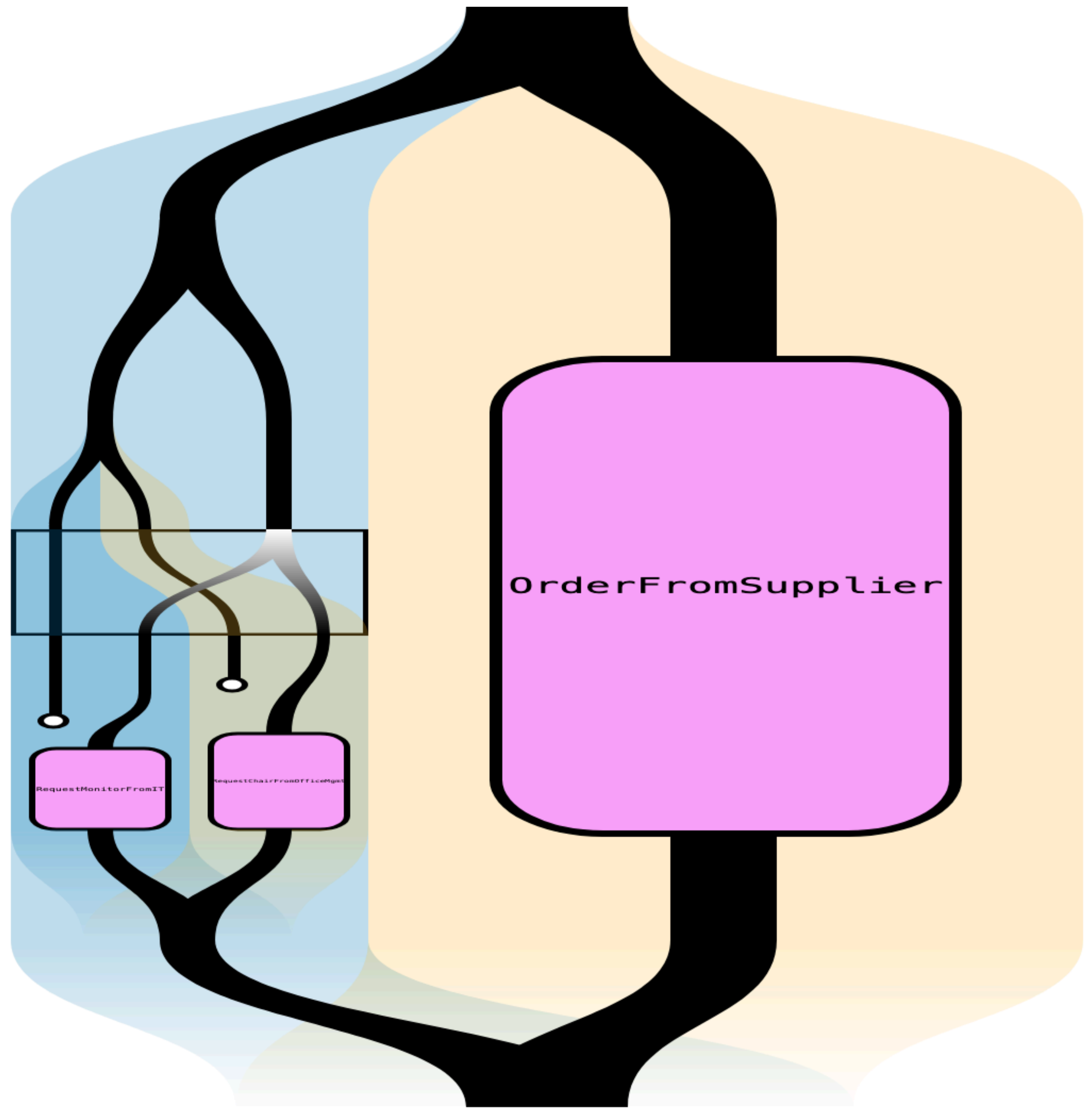
1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed
7. Lower from Flow^1 to Flow



`switch:` What Does It *Do*?

```
req switch (
  is { case ForOffice(Monitor(_) ** deskLoc) => requestMonitorFromIT(deskLoc)        },
  is { case ForOffice(Chair(_) ** deskLoc)   => requestChairFromOfficeMgmt(deskLoc) },
  is { case WorkFromHome(item ** address)    => orderFromSupplier(item ** address)  }
)
```

1. Delambdify each case
2. Pick the first Extractor,
   obtain the whole partitioning
3. Group by partition
   empty group = non-exhaustivity
4. Remove the matched extractor
5. Apply 2.-6. inside each group
6. Construct Handlers,
   distribute as needed
7. Lower from Flow[1] to Flow



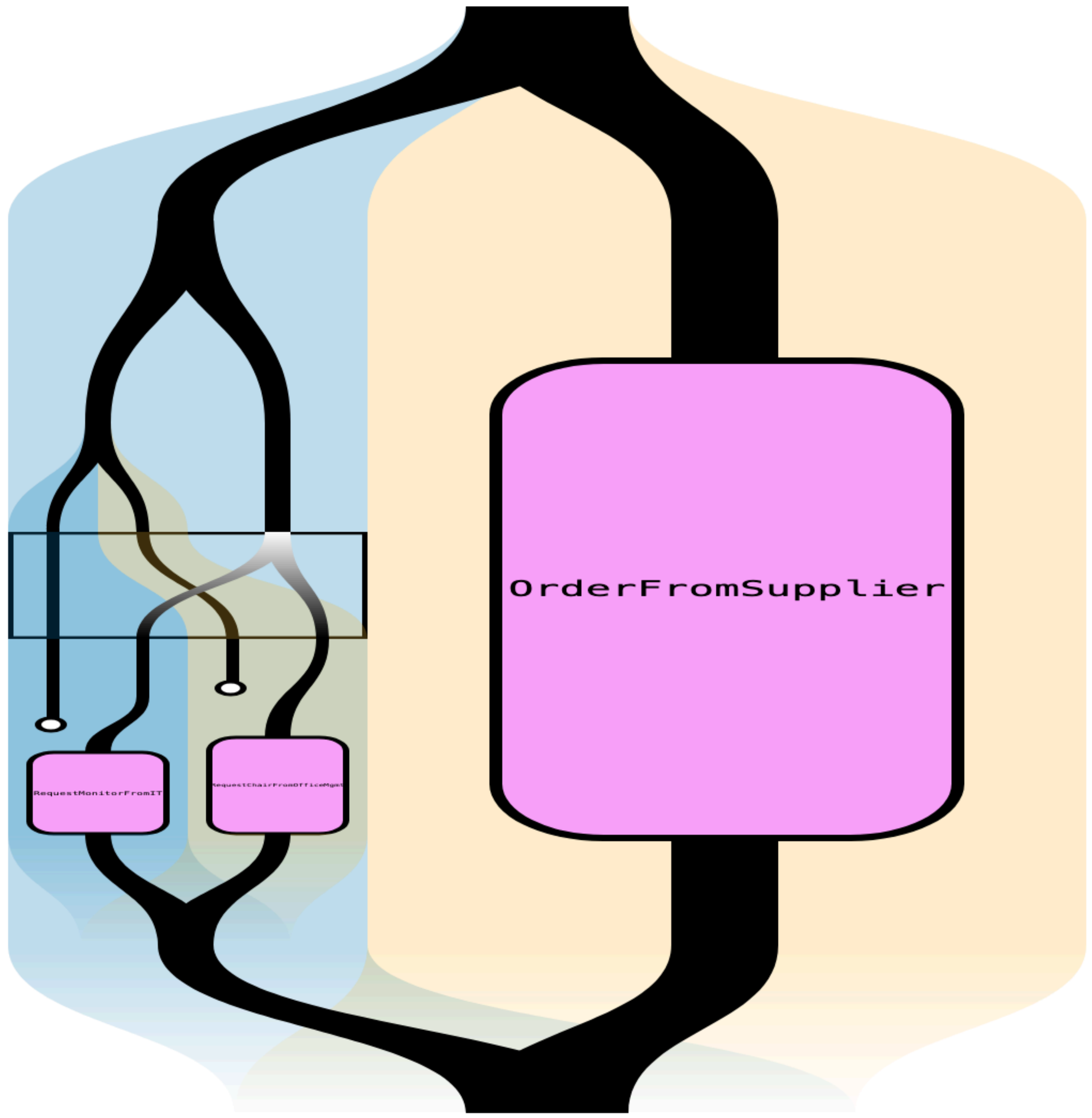
```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do?
3. What do the **extractors** do? ✔
   (`ForOffice`, `Monitor`, ...)

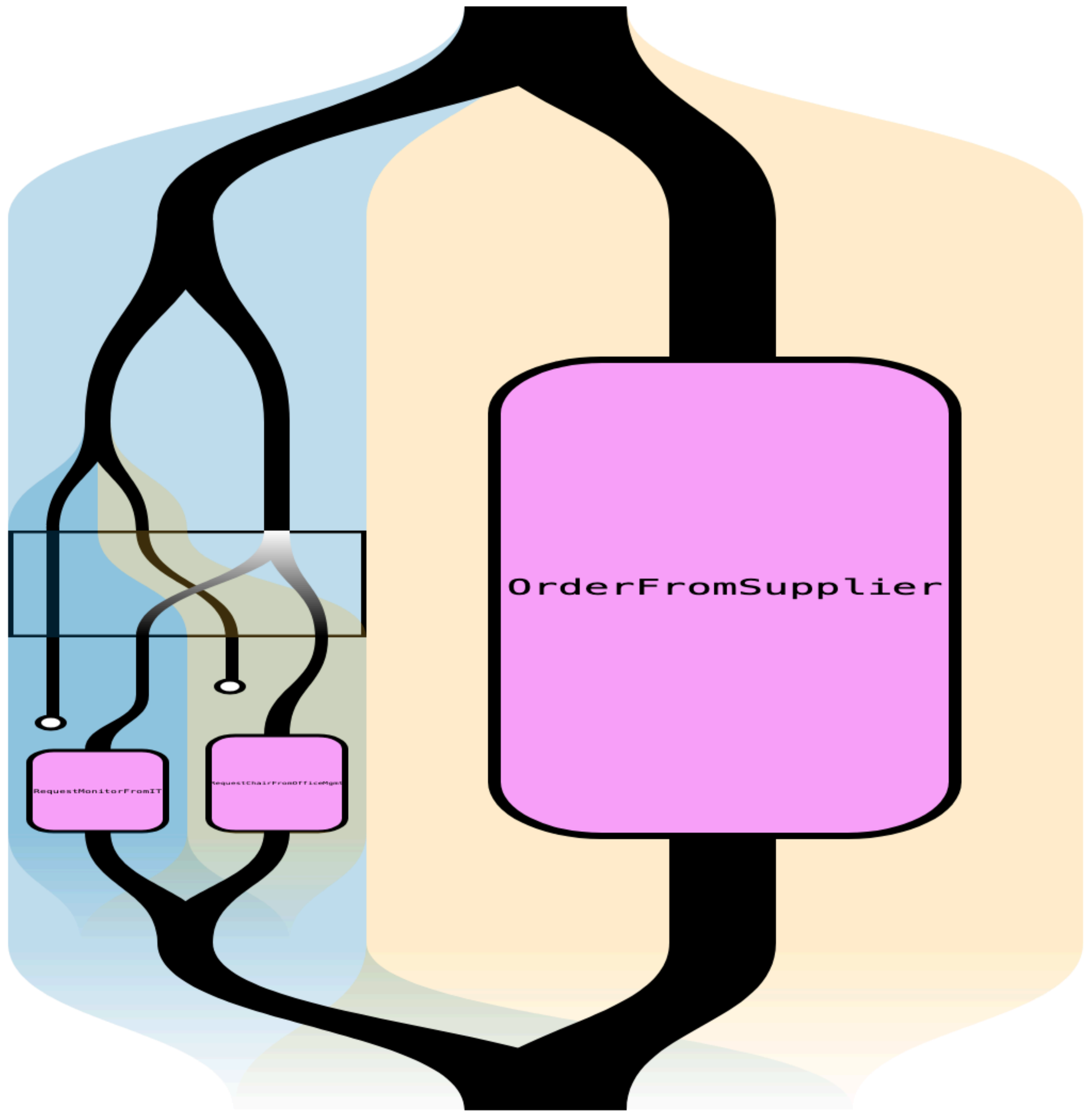
```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

1. What does **`Flow`** do? ✔
2. What does **`switch`** do? ✔
3. What do the **extractors** do? ✔
   (`ForOffice`, `Monitor`, …)

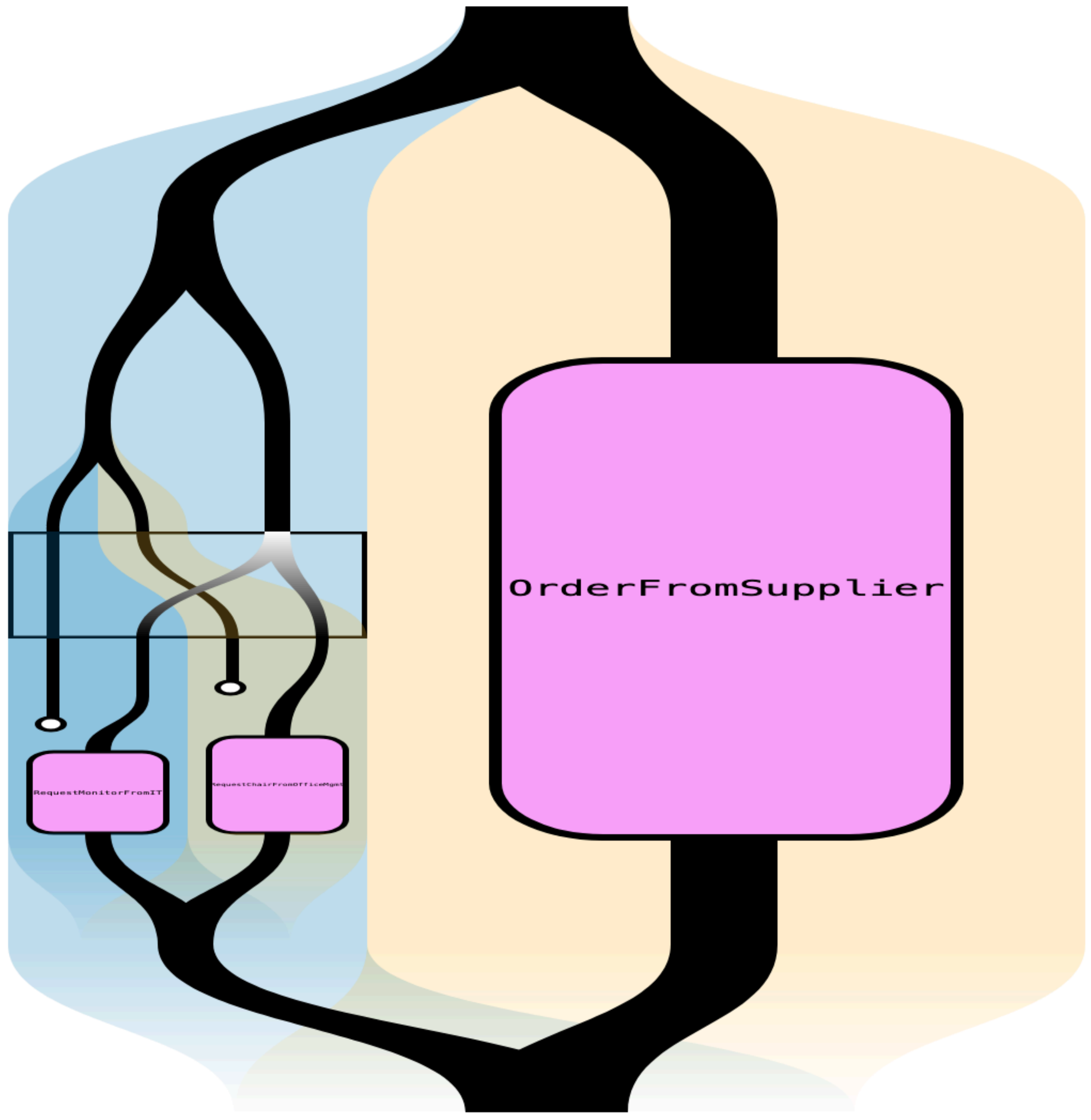
```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```



```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

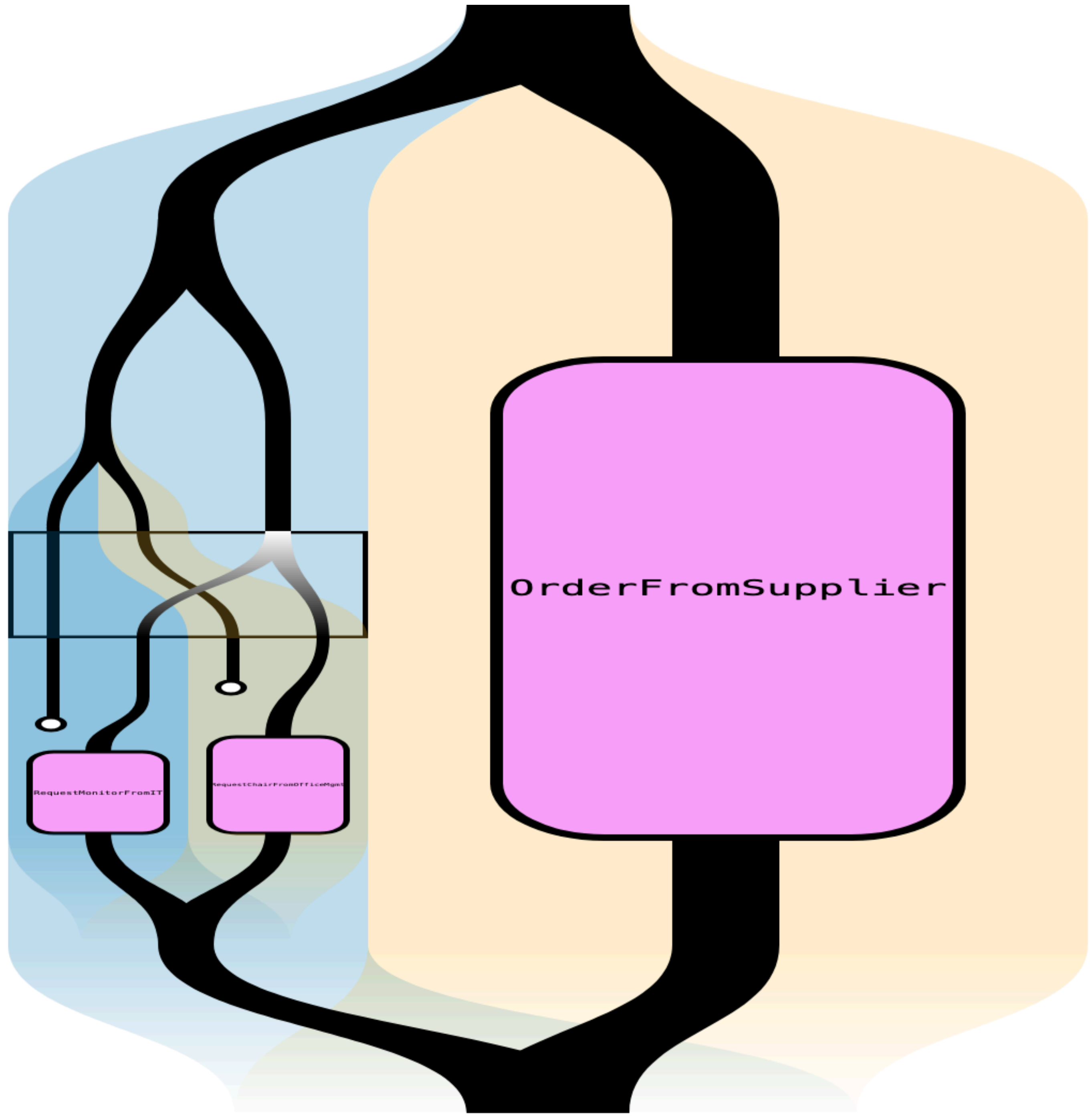


```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```



✔ Compiled Scala-like pattern matching

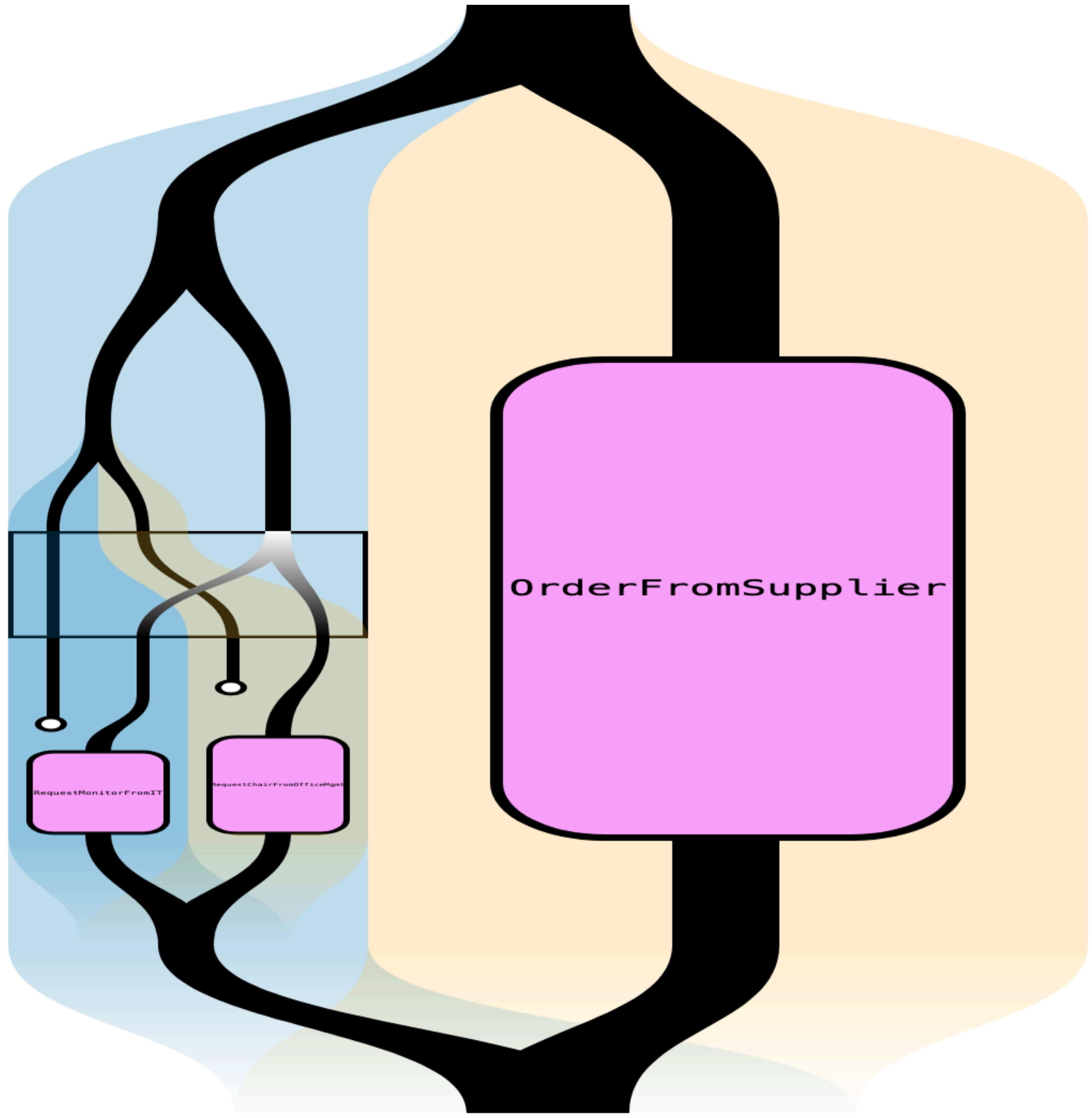
```scala
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```



✔ Compiled Scala-like pattern matching

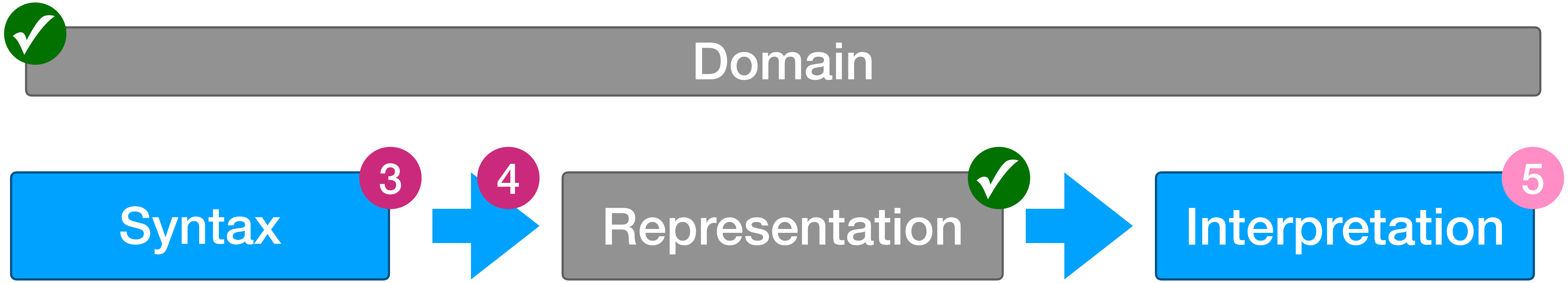
✔ Representation **exhaustive by construction**

```
Flow { req =>
  req switch {
    case ForOffice(Monitor(_) ** deskLoc) =>
      requestMonitorFromIT(deskLoc)
    case ForOffice(Chair(_) ** deskLoc) =>
      requestChairFromOfficeMgmt(deskLoc)
    case WorkFromHome(item ** address) =>
      orderFromSupplier(item ** address)
  }
}
```

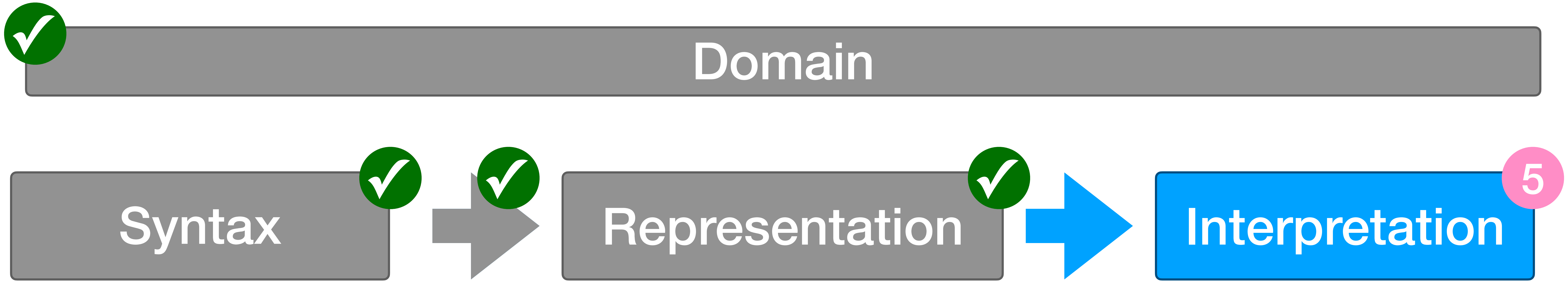


- ✔ Compiled Scala-like pattern matching
- ✔ Representation **exhaustive by construction**
- ✔ Non-exhaustivity reported in **embedded compile-time**

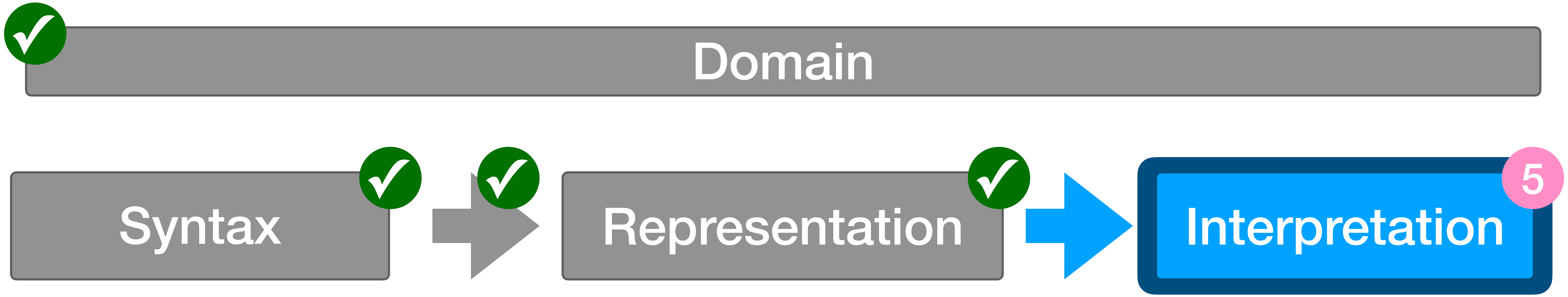
Agenda



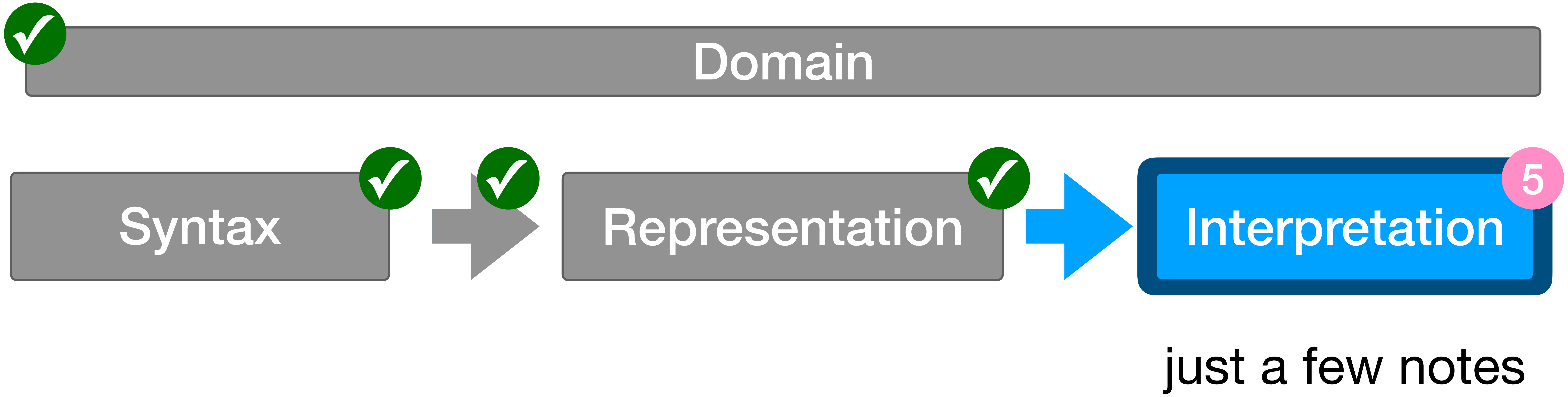
Agenda



Agenda



Agenda


Domain
Syntax
Representation
Interpretation
5
just a few notes

Interpretation: What’s Possible

Interpretation: What’s Possible

- Durable execution

Interpretation: What's Possible

- Durable execution
    - Workflow is pure data

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data

Interpretation: What’s Possible

- Durable execution
    - Workflow is pure data
    - Workflow-in-progress is pure data
    - No *need* for event sourcing

Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization

Interpretation: What’s Possible

- Durable execution
    - Workflow is pure data
    - Workflow-in-progress is pure data
    - No *need* for event sourcing
- Visualization
    - at arbitrary level of detail

Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads

Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input

Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state
- Workflow Templates

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state
- Workflow Templates
  - just a regular Workflow

Interpretation: What’s Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state
- Workflow Templates
  - just a regular Workflow
  - instantiated by partial application

Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state
- Workflow Templates
  - just a regular Workflow
  - instantiated by partial application
- Interactive Dry Run

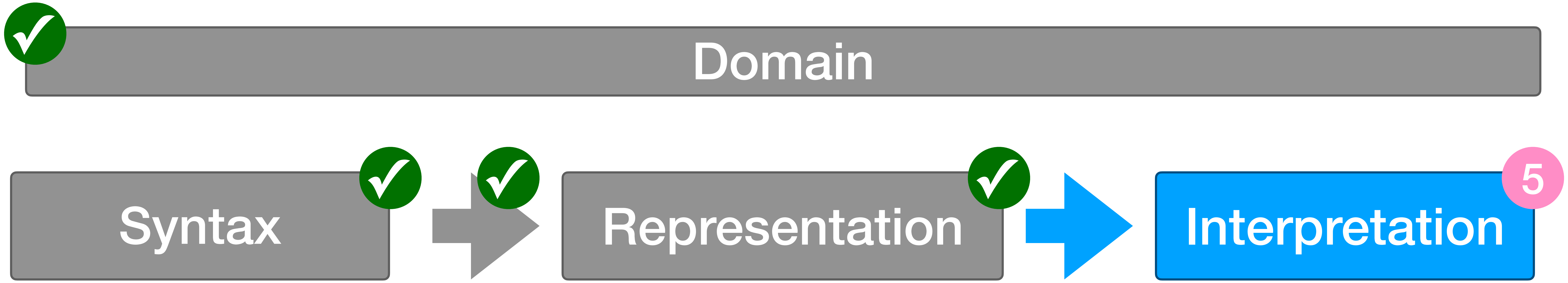
Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state
- Workflow Templates
  - just a regular Workflow
  - instantiated by partial application
- Interactive Dry Run
- Graphical Designer / IDE

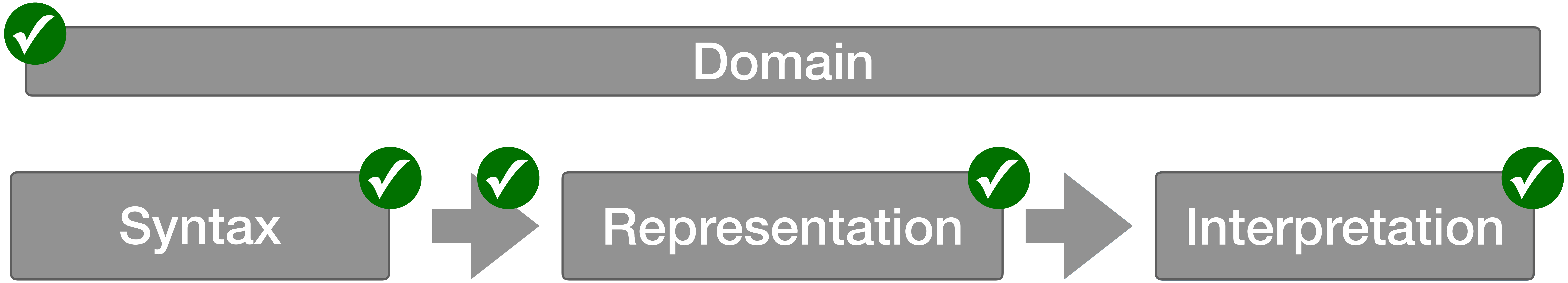
Interpretation: What's Possible

- Durable execution
  - Workflow is pure data
  - Workflow-in-progress is pure data
  - No *need* for event sourcing
- Visualization
  - at arbitrary level of detail
  - incl. mid-execution
- Concurrency without threads
- Cancellation without threads
- Commands/Queries as regular input
  - of some Stream type
  - no need for mutable state
- Workflow Templates
  - just a regular Workflow
  - instantiated by partial application
- Interactive Dry Run
- Graphical Designer / IDE
  - as an alternative to the Scala embedding

Agenda



Agenda



Take Aways

Take Aways

Shallow embedding has **hard limits.**

Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

More than one way to **represent functions.**

Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

More than one way to **represent functions.**

Some better at **preventing illegal programs** than others.

Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

More than one way to **represent functions.**

Some better at **preventing illegal programs** than others.

Even **exhaustivity by construction** is possible.

Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

More than one way to **represent functions.**

Some better at **preventing illegal programs** than others.

Even **exhaustivity by construction** is possible.

https://github.com/TomasMikula/libretto/…/lambda-examples/…/workflow

Take Aways

Shallow embedding has **hard limits.**

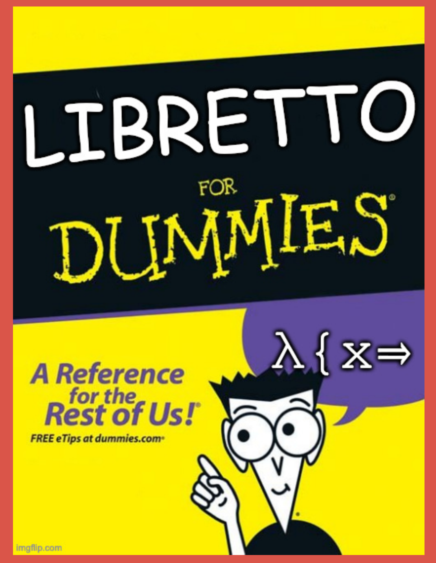
Regain **control** by going **deep.**

More than one way to **represent functions.**

Some better at **preventing illegal programs** than others.

Even **exhaustivity by construction** is possible.

https://github.com/TomasMikula/libretto/…/lambda-examples/…/workflow



Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

More than one way to **represent functions.**

Some better at **preventing illegal programs** than others.

Even **exhaustivity by construction** is possible.

https://github.com/TomasMikula/libretto/…/lambda-examples/…/workflow

Take Aways

Shallow embedding has **hard limits.**

Regain **control** by going **deep.**

More than one way to **represent functions.**

Some better at **preventing illegal programs** than others.

Even **exhaustivity by construction** is possible.

https://github.com/TomasMikula/libretto/…/lambda-examples/…/workflow