When Your DSL Needs to Support User-Defined *Domain* Functions

Affordable Deep Embeddings via Curry-Howard-*Lambek* Correspondence

Scala Days MADRID

September 14th, 2023

Tomas Mikula

What This Talk Is About

What This Talk Is About

DSLs

What This Talk Is About

DSLs

external

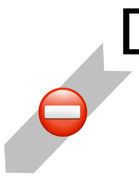
What This Talk Is About

DSLs

external

- Custom syntax
- Specialized tools
- Good error messages
- Build your own
  - Parser
  - Type checker
  - IDE integration

What This Talk Is About



- Custom syntax
- Specialized tools
- Good error messages
- Build your own
  - Parser
  - Type checker
  - IDE integration

What This Talk Is About

DSLs → embedded (internal)

external

What This Talk Is About

DSLs → embedded (internal)

external

- Constrained syntax
- Not so good error messages
- No custom IDE
- Piggy-back on host language's
  - Parser
  - Type checker
  - IDE integration

What This Talk Is About

DSLs → embedded (internal)

external

What This Talk Is About

DSLs

external

embedded
(internal)

fluent syntax for
imperative interface

What This Talk Is About

DSLs

external

embedded
(internal)

fluent syntax for
imperative interface

What This Talk Is About

DSLs

external

embedded
(internal)

fluent syntax for
imperative interface

programs
as values

What This Talk Is About

DSLs

external

embedded
(internal)

fluent syntax for
imperative interface

programs
as values

shallowly
embedded
Scala functions
inside

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded

Scala functions inside

For example

- Parser combinators
- “Monadic” DSLs with deferred evaluation

- Easy to make
- Not introspectable
- Single interpretation
  - i.e. *run* them

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded

Scala functions inside

For example

- Parser combinators
- “Monadic” DSLs with deferred evaluation

- Easy to make
- Not introspectable
- Single interpretation
  - i.e. *run* them

What This Talk Is About

DSLs

external

embedded
(internal)

fluent syntax for
imperative interface

programs
as values

shallowly
embedded
Scala functions
inside

introspectable AST
(programs as data)

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

programs as values

introspectable AST
(programs as data)

- Alternative interpretations
  - Serialize
  - Visualize
  - Statically analyze
  - Generate code from
  - …

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded
Scala functions inside

introspectable AST
(programs as data)

no user-defined functions
(or severely limited)

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded
Scala functions inside

introspectable AST
(programs as data)

no user-defined functions
(or severely limited)

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded
Scala functions inside

introspectable AST
(programs as data)

user-defined functions

no user-defined functions
(or severely limited)

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded
Scala functions inside

introspectable AST
(programs as data)

user-defined functions

point-free
notation **only**

no user-defined functions
(or severely limited)

W alk Is About

point-full

```
x => g(f(x))


_ match {
  case Left(a)  => f(a)
  case Right(b) => g(b)
}
```

DSLs

ent syntax for
erative interface

external

shallowly
embedded
Scala functions
inside

programs
as values

introspectable AST
(programs as data)

user-defined
functions

no user-defined
functions
(or severely limited)

point-free
notation **only**

W a out

point-full

```
x => g(f(x))


_ match {
  case Left(a)  => f(a)
  case Right(b) => g(b)
}
```

point-free

```
f andThen g


Either.fold(f, g)
```

DSLs

external

shallowly
mbedded
cala functions
inside

rospectable AST
(programs as data)

user-defined
functions

point-free
notation **only**

no user-defined
functions
(or severely limited)

W a out

point-full

```
x => g(f(x))


_ match {
  case Left(a)  => f(a)
  case Right(b) => g(b)
}
```

point-free

```
f andThen g


Either.fold(f, g)
```

DSLs

external

shallowly
mbedded
cala functions
inside

rospectable AST
(programs as data)

- Define a function without giving a name to its input
- Easy to represent as data
- Good for programs
- Hard for humans

user-defined
functions

no user-defined
functions
(or severely limited)

point-free
notation **only**

W a out

DSLs

external

shallowly
mbedded
cala functions
inside

point-full

```
x => g(f(x))

_ match {
  case Left(a)  => f(a)
  case Right(b) => g(b)
}
```

point-free

```
f andThen g


Either.fold(f, g)
```

rospectable AST
(programs as data)

- Define a function without giving a name to its input
- Easy to represent as data
- Good for programs
- Hard for humans

user-defined
functions

no user-defined
functions
(or severely limited)

point-free
notation **only**

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded
Scala functions inside

introspectable AST
(programs as data)

supports variables

user-defined functions

point-free notation **only**

no user-defined functions
(or severely limited)

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

programs as values

introspectable AST
(programs as data)

no user-defined functions
(or severely limited)

user-defined functions

supports variables

quoted DSLs

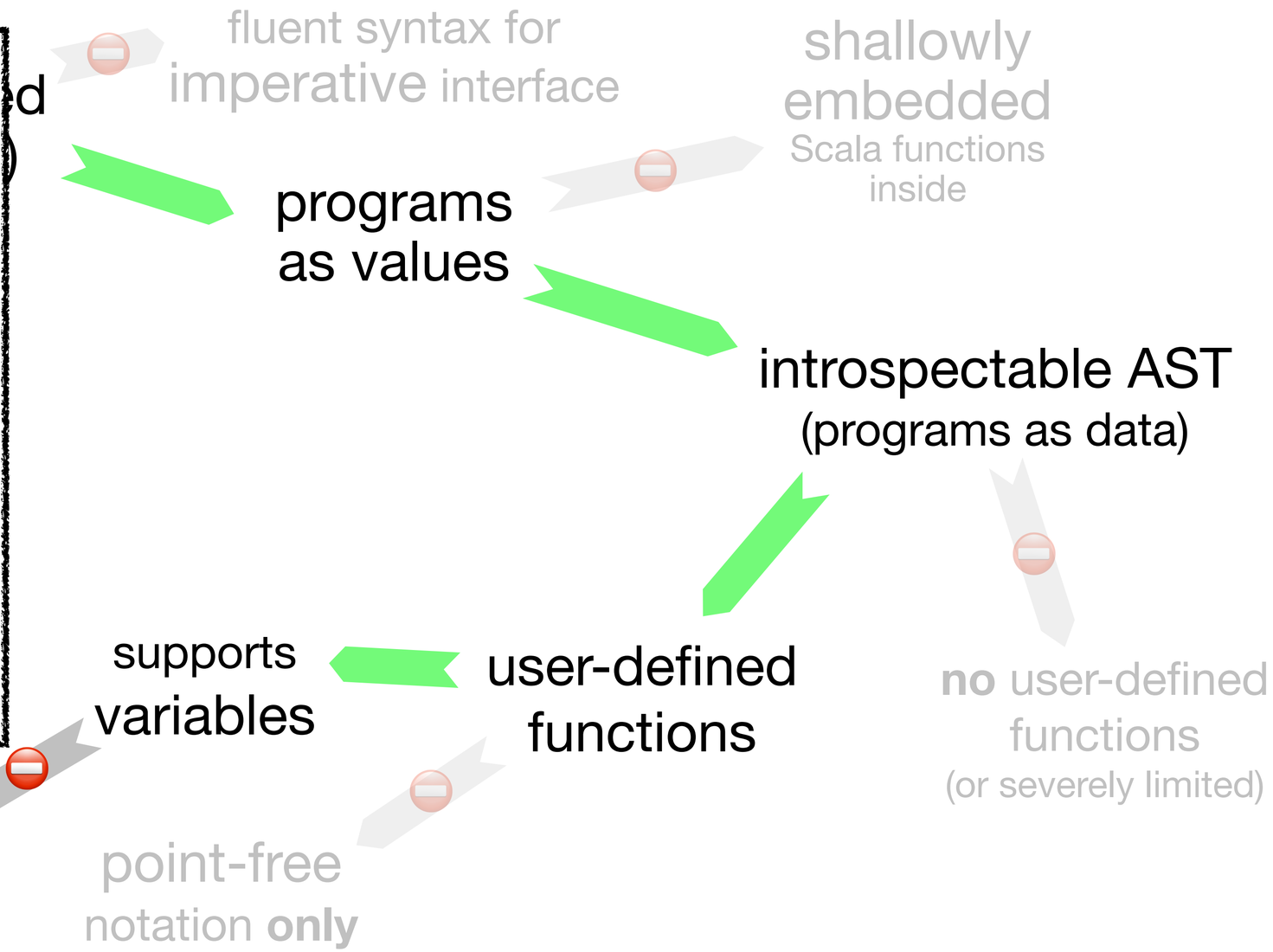
point-free notation **only**

What This Talk Is About

- Use Scala's anonymous function syntax
- Use a macro to give it a different meaning
- Dealing with Scala AST
- Macro does not see through an `Ident`

What This Talk Is About

- Use Scala's anonymous function syntax
- Use a macro to give it a different meaning
- Dealing with Scala AST
- Macro does not see through an `Ident`



What This Talk Is About

DSLs

embedded (internal)

fluent syntax for imperative interface

external

shallowly embedded
Scala functions inside

programs as values

introspectable AST
(programs as data)

avoid dealing with Scala AST

supports variables

user-defined functions

no user-defined functions
(or severely limited)

quoted DSLs

point-free notation **only**

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

programs as values

introspectable AST
(programs as data)

no user-defined functions
(or severely limited)

user-defined functions

supports variables

point-free
notation **only**

quoted
DSLs

avoid dealing with Scala AST

give up Scala's variable binding

What This Talk Is About

fluent syntax for

shallowly

ded

ctions

e

DSLs

external

```
// Declare your aliases before using them in SQL:
Author a = AUTHOR.as("a");

// Use aliased tables in your statement
create.select()
      .from(a)
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")));
```

ctable AST

ms as data)

give up Scala's
variable binding

avoid dealing
with Scala AST

supports
variables

user-defined
functions

no user-defined
functions
(or severely limited)

quoted
DSLs

point-free
notation **only**

What This Talk Is About

fluent syntax for

shallowly
ded
ctions

DSLs

external

```
// Declare your aliases before using them in SQL:
Author a = AUTHOR.as("a");

// Use aliased tables in your statement
create.select()
      .from(a)
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")));
```

ctable AST
ms as data)

give up Scala's
variable binding

avoid dealing
with Scala AST

supports
variables

user-defined
functions

no user-defined
functions
(or severely limited)

quoted
DSLs

point-free
notation **only**

What This Talk Is About

fluent syntax for

shallowly
ded
ctions

DSLs

external

```
// Declare your aliases before using them in SQL:
Author a = AUTHOR.as("a");

// Use aliased tables in your statement
create.select()
      .from(a)
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")));
```

ctable AST
ms as data)

give up Scala's
variable binding

avoid dealing
with Scala AST

supports
variables

user-defined
functions

no user-defined
functions
(or severely limited)

quoted
DSLs

point-free
notation **only**

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

programs as values

shallowly embedded
Scala functions inside

introspectable AST
(programs as data)

user-defined functions

no user-defined functions
(or severely limited)

supports variables

point-free
notation **only**

avoid dealing with Scala AST

quoted
DSLs

give up Scala's variable binding

What This Talk Is About

DSLs

external

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

programs as values

introspectable AST
(programs as data)

user-defined functions

no user-defined functions
(or severely limited)

supports variables

point-free
notation **only**

avoid dealing with Scala AST

quoted
DSLs

give up Scala's variable binding

What This Talk Is About

DSLs

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

external

programs as values

introspectable AST
(programs as data)

delambdify

give up Scala's variable binding

avoid dealing with Scala AST

supports variables

user-defined functions

no user-defined functions
(or severely limited)

quoted DSLs

point-free notation **only**

What This

Instead of

DSLs

embedded
(internal)

external

delambdify

give up Scala's
variable binding

avoid dealing
with Scala AST

quoted
DSLs

notation **only**

What This

DSLs

embedded (internal)

external

delambdify

give up Scala's variable binding

avoid dealing with Scala AST

quoted DSLs

notation **only**

Instead of

```
(a: A) => body: B
```

What This

DSLs

embedded (internal)

external

give up Scala's variable binding

delambdify

avoid dealing with Scala AST

quoted DSLs

notation **only**

Instead of

```
(a: A) => body: B
```

we write

What This

DSLs

embedded (internal)

external

delambdify

give up Scala's variable binding

avoid dealing with Scala AST

quoted DSLs

notation **only**

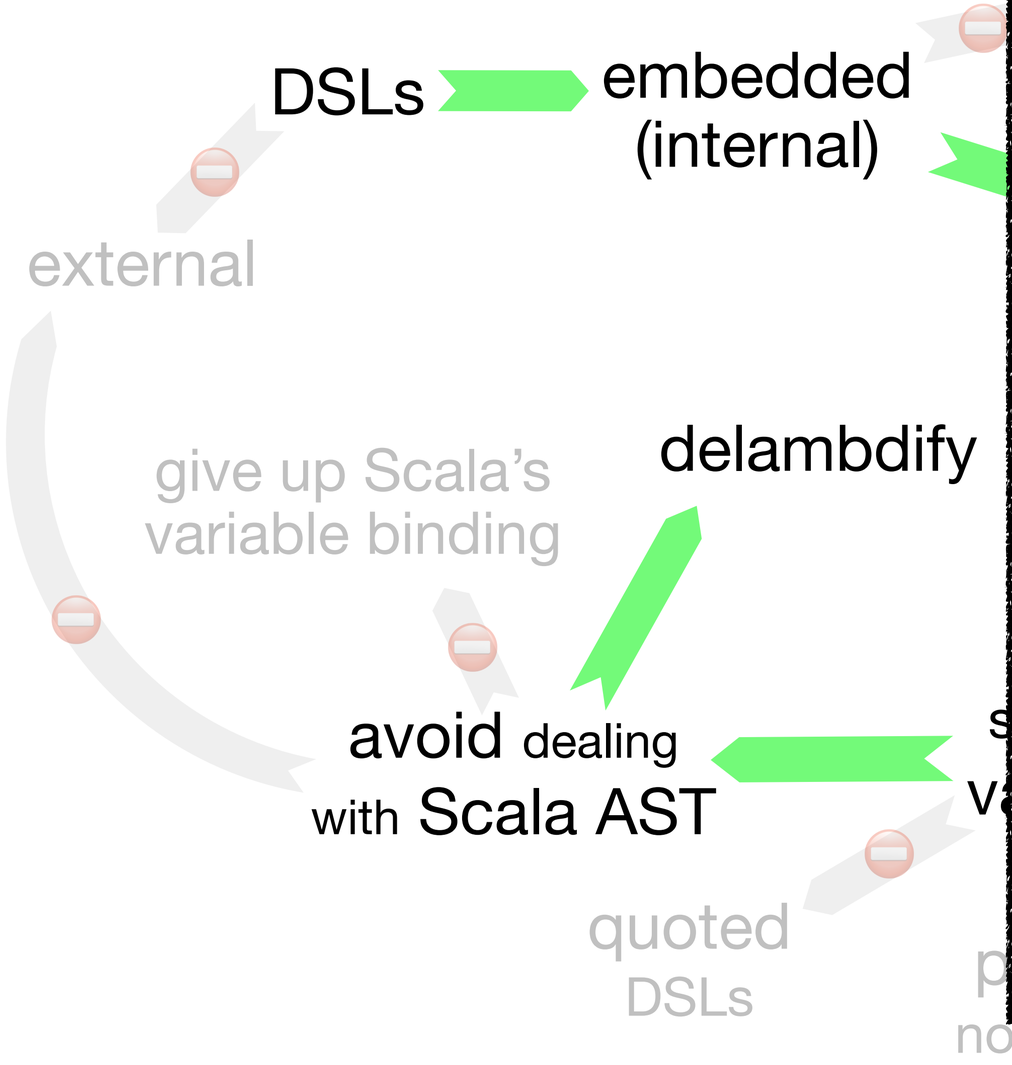
Instead of

```
(a: A) => body: B
```

we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

What This



Instead of

```
(a: A) => body: B
```

we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

which is converted to

What This

DSLs

embedded (internal)

external

give up Scala's variable binding

delambdify

avoid dealing with Scala AST

quoted DSLs

notation **only**

Instead of

```
(a: A) => body: B
```

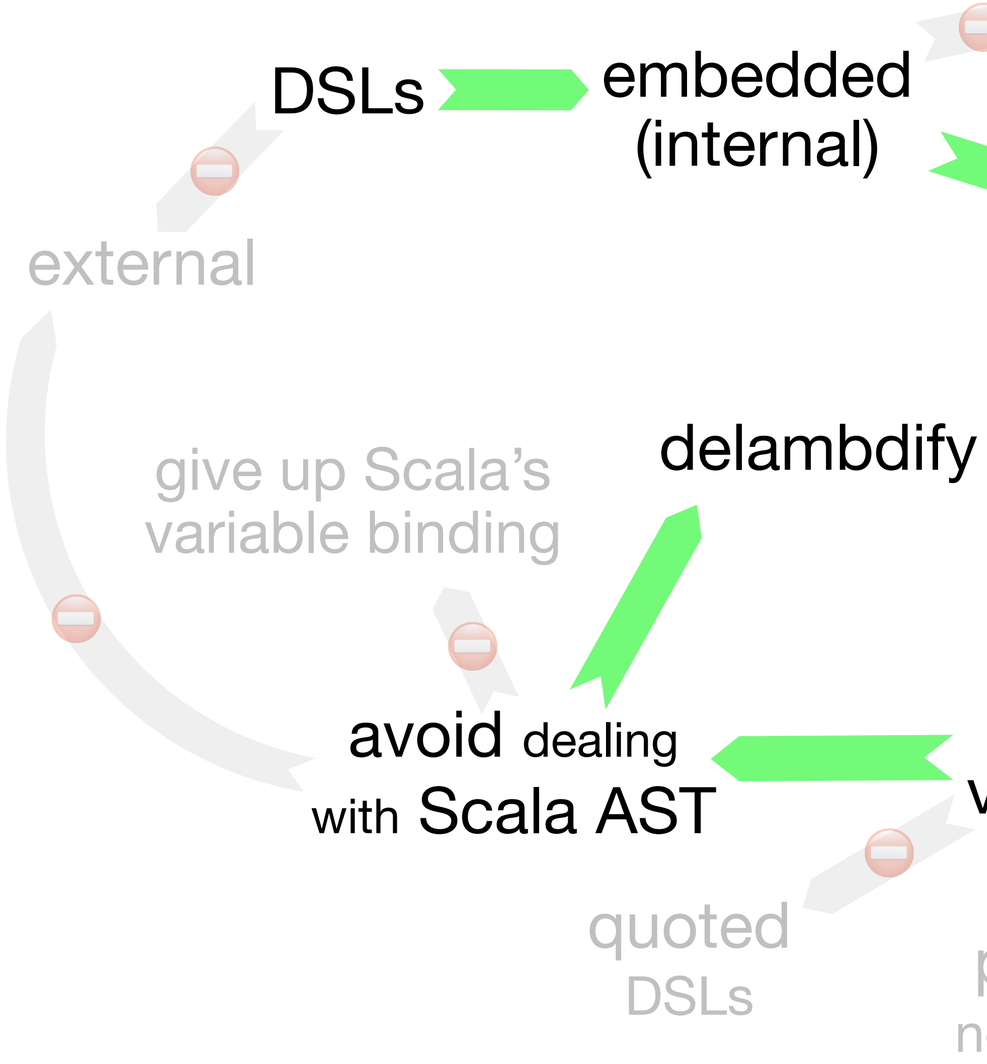
we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

which is converted to

```
Lambda("a", body): Fun[A, B]
```

What This



Instead of

```
(a: A) => body: B
```

we write

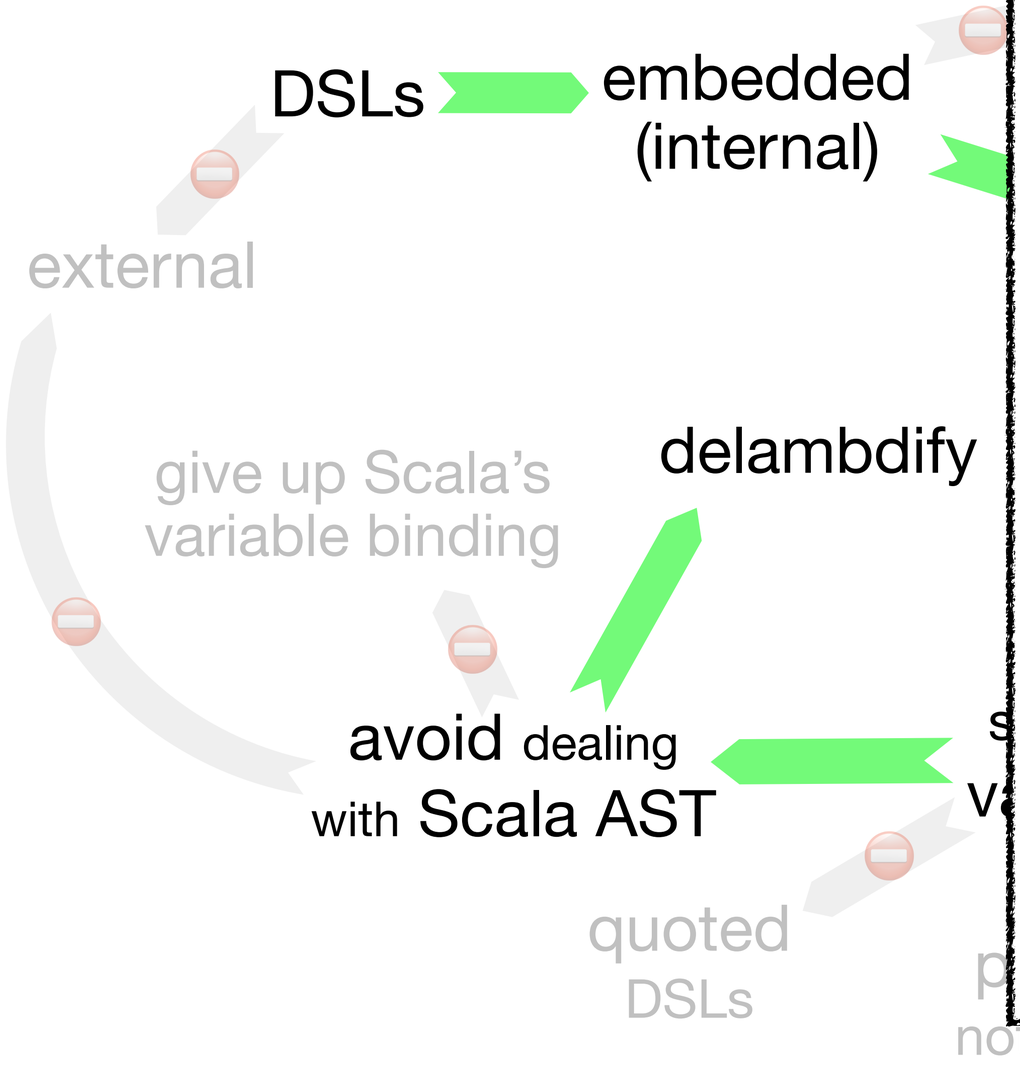
```
fun { (a: Expr[A]) => body: Expr[B] }
```

which is converted to

```
Lambda("a", body): Fun[A, B]
```

where `body` contains occurrences of `Var("a")`.

What This



Instead of

```
(a: A) => body: B
```

we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

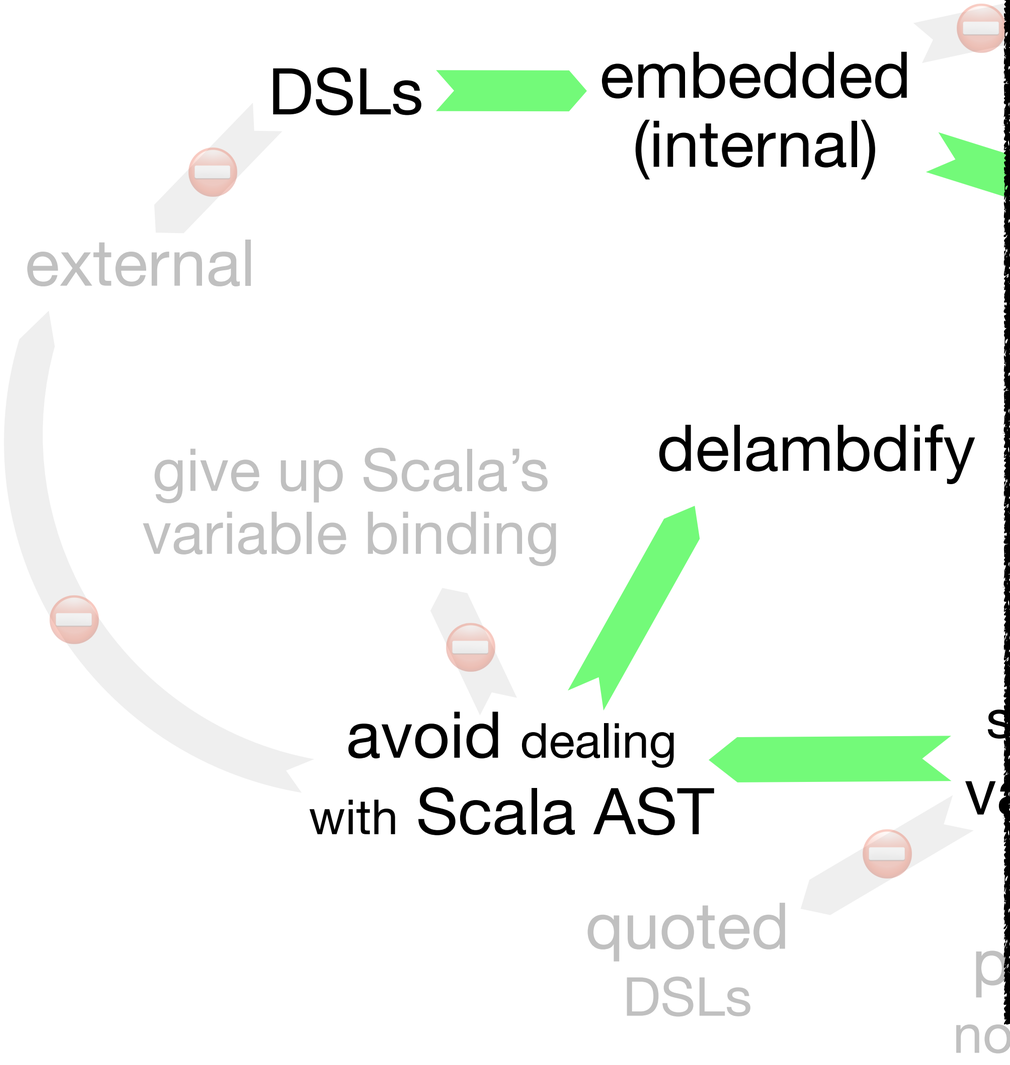
which is converted to

```
Lambda("a", body): Fun[A, B]
```

where `body` contains occurrences of `Var("a")`.

- Programs as data

What This



Instead of

```
(a: A) => body: B
```

we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

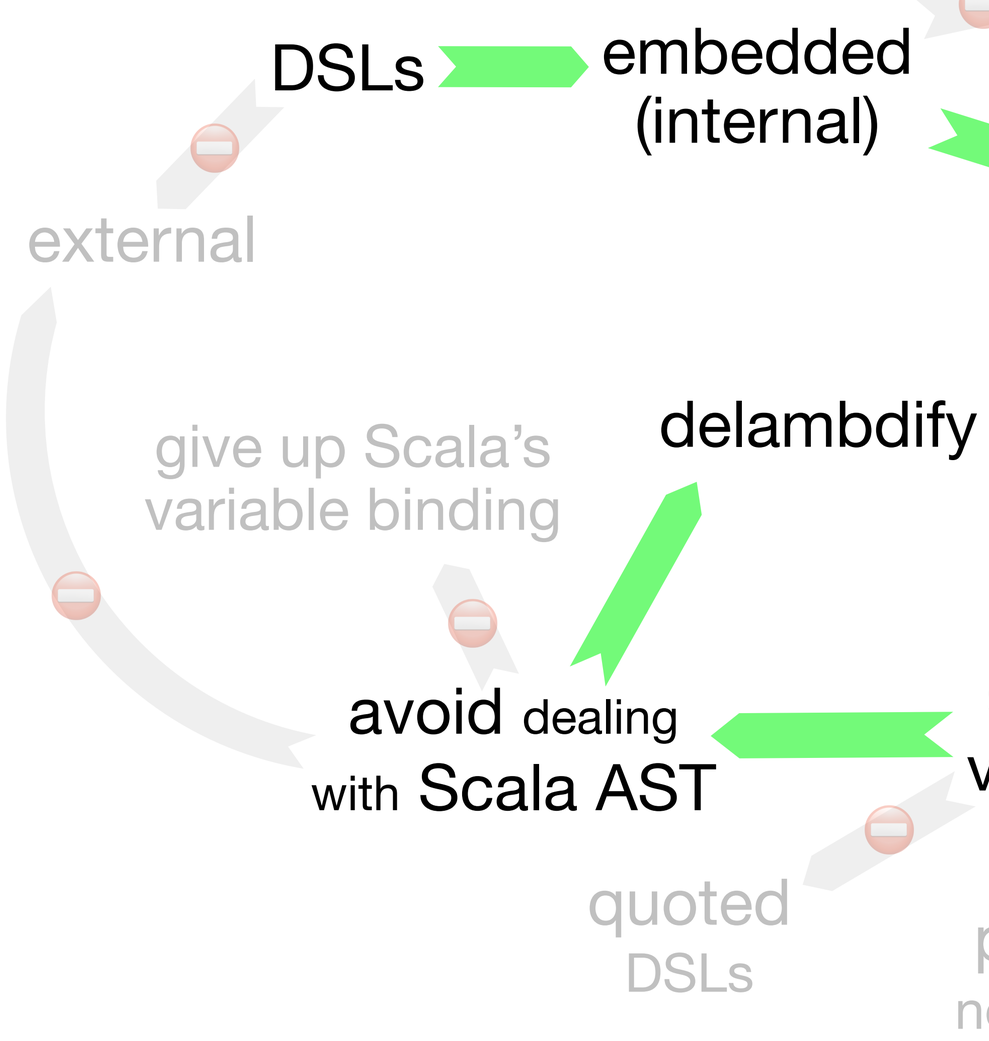
which is converted to

```
Lambda("a", body): Fun[A, B]
```

where `body` contains occurrences of `Var("a")`.

- Programs as data
- Dealing with variables error-prone

What This



Instead of

```
(a: A) => body: B
```

we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

which is converted to

```
Lambda("a", body): Fun[A, B]
```

where `body` contains occurrences of `Var("a")`.

- Programs as data
- Dealing with variables error-prone
- Non-locality of substitution

What This

DSLs

embedded (internal)

external

give up Scala's variable binding

delambdify

avoid dealing with Scala AST

quoted DSLs

notation **only**

Instead of

```
(a: A) => body: B
```

we write

```
fun { (a: Expr[A]) => body: Expr[B] }
```

which is converted to

```
Lambda("a", body): Fun[A, B]
```

where `body` contains occurrences of `Var("a")`.

- Programs as data
- Dealing with variables error-prone
- Non-locality of substitution
- Malformed programs representable

What This Talk Is About

DSLs

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

external

programs as values

introspectable AST
(programs as data)

delambdify

give up Scala's variable binding

avoid dealing with Scala AST

supports variables

user-defined functions

no user-defined functions
(or severely limited)

quoted DSLs

point-free notation **only**

What This Talk Is About

DSLs

embedded (internal)

fluent syntax for imperative interface

shallowly embedded
Scala functions inside

external

programs as values

introspectable AST
(programs as data)

convert to point-free representation

delambdify

give up Scala's variable binding

avoid dealing with Scala AST

supports variables

user-defined functions

no user-defined functions
(or severely limited)

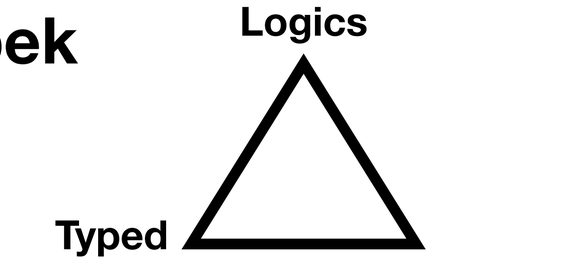
quoted DSLs

point-free notation **only**

What This Talk Is About

Curry-Howard-Lambek

correspondence



DSLs

embe
(inte

external

introspectable AST
(programs as data)

convert to
point-free
representation

delambdify

give up Scala's
variable binding

avoid dealing
with Scala AST

supports
variables

user-defined
functions

no user-defined
functions
(or severely limited)

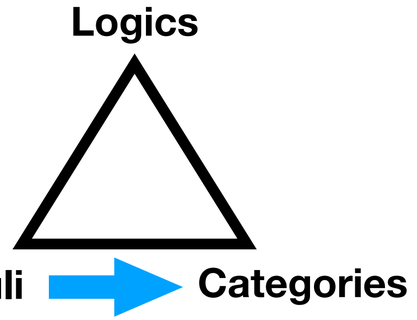
quoted
DSLs

point-free
notation **only**

What This Talk Is About

Curry-Howard-Lambek

correspondence



DSLs

embe
(inte

external

give up Scala's
variable binding

delambdify

convert to
point-free
representation

introspectable AST
(programs as data)

avoid dealing
with Scala AST

supports
variables

user-defined
functions

no user-defined
functions
(or severely limited)

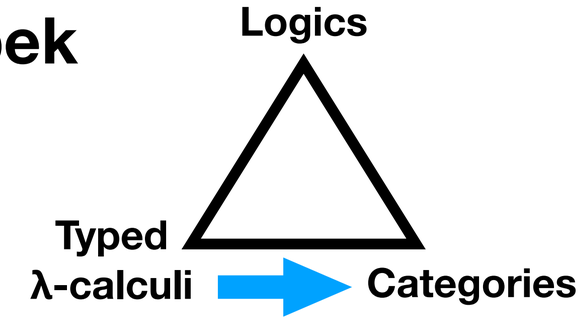
quoted
DSLs

point-free
notation **only**

What This Talk Is About

Curry-Howard-Lambek

correspondence



DSLs

embe
(inte

external

introspectable AST
(programs as data)

convert to
point-free
representation

delambdify

give up Scala's
variable binding

avoid dealing
with Scala AST

sup
var

- Users get variables and
- lambda syntax

user-defined
functions
severely limited)

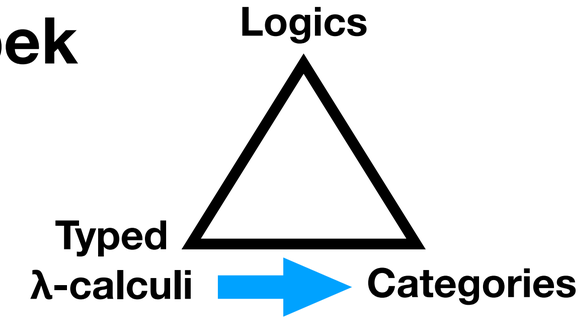
quoted
DSLs

po
nota

What This Talk Is About

Curry-Howard-Lambek

correspondence



DSLs

embe
(inte

external

introspectable AST
(programs as data)

convert to
point-free
representation

delambdify

give up Scala's
variable binding

avoid dealing
with Scala AST

sup
var

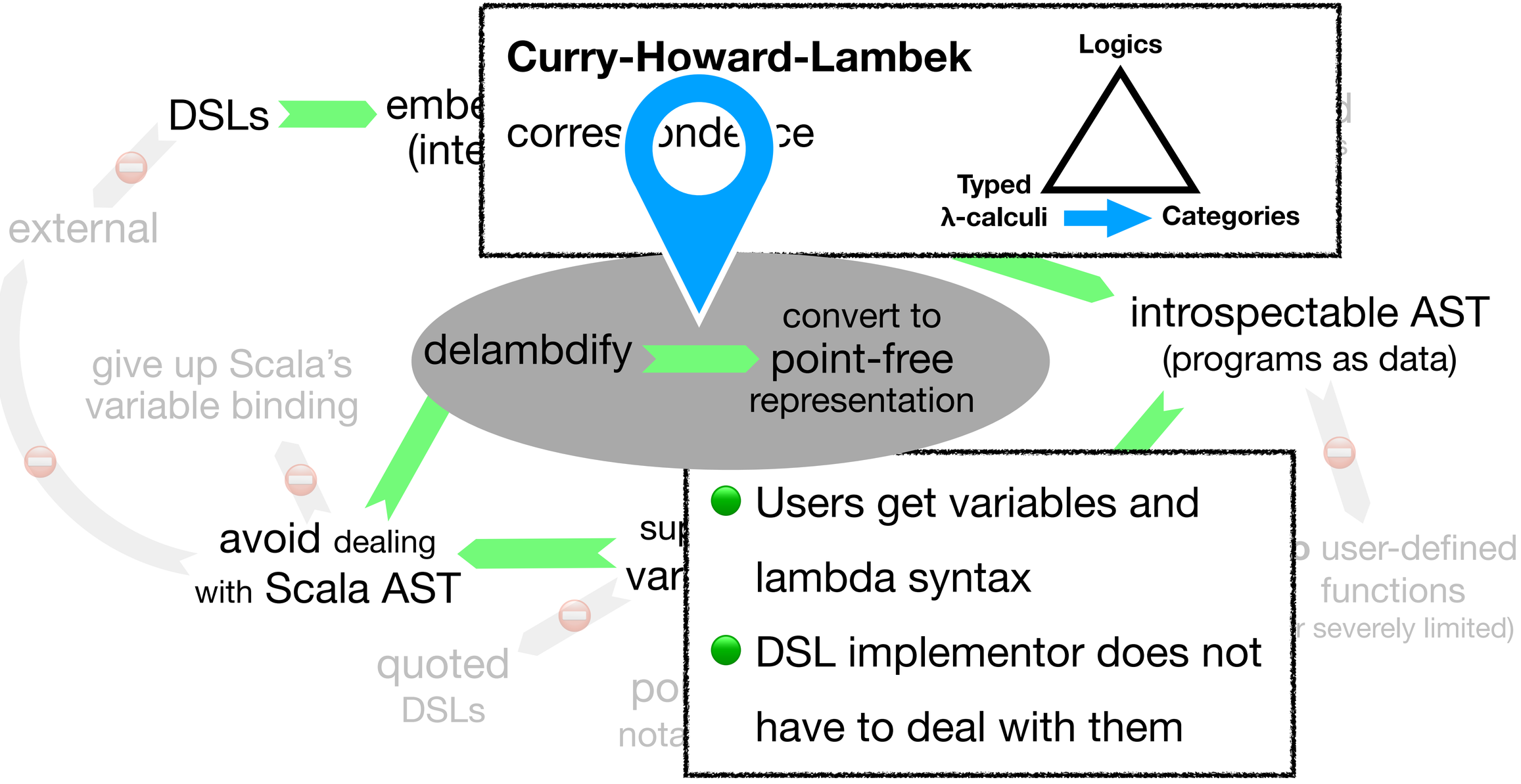
- Users get variables and lambda syntax
- DSL implementor does not have to deal with them

quoted
DSLs

po
nota

user-defined
functions
severely limited)

What This Talk Is About



Goal

Language User

writes lambdas

```
fun { a =>
  val b = f(a)
  val c = g(a)
  h(b, c)
}
```

Goal

Language User

writes lambdas

```
fun { a =>
  val b = f(a)
  val c = g(a)
  h(b, c)
}
```

Language Developer

consumes data structures

Goal

Language User

writes lambdas

```
fun { a =>
  val b = f(a)
  val c = g(a)
  h(b, c)
}
```

Language Developer

consumes data structures

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    h
)): AST[A, B]
```

Goal

Language User

writes lambdas

```
fun { a =>
  val b = f(a)
  val c = g(a)
  h(b, c)
}
```

Language Developer

consumes data structures

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    h
)): AST[A, B]
```

```
enum AST[A, B]:
  // Pure data!
  // No Scala functions
  // inside!

  case AndThen[A, B, C](
    f: AST[A, B],
    g: AST[B, C],
  ) extends AST[A, C]

  case Par[A1, A2, B1, B2](
    f1: AST[A1, B1],
    f2: AST[A2, B2],
  ) extends AST[(A1, A2), (B1, B2)]

  case Dup[A]() extends AST[A, (A, A)]

  // ...
```

Goal

Language User

writes lambdas

```
fun { a =>
  val b = f(a)
  val c = g(a)
  h(b, c)
}
```


Language Developer

consumes data structures

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    h
)): AST[A, B]
```

```
enum AST[A, B]:
  // Pure data!
  // No Scala functions
  // inside!

  case AndThen[A, B, C](
    f: AST[A, B],
    g: AST[B, C],
  ) extends AST[A, C]

  case Par[A1, A2, B1, B2](
    f1: AST[A1, B1],
    f2: AST[A2, B2],
  ) extends AST[(A1, A2), (B1, B2)]

  case Dup[A]() extends AST[A, (A, A)]

  // ...
```

Goal

Language User

writes lambdas

```
fun { a =>
  val b = f(a)
  val c = g(a)
  h(b, c)
}
```

provided by the
Libretto library

Language Developer

consumes data structures

```
AndThen(
  Dup(),
  AndThen(
    Par(f, g),
    h
)): AST[A, B]
```

```
enum AST[A, B]:
  // Pure data!
  // No Scala functions
  // inside!

  case AndThen[A, B, C](
    f: AST[A, B],
    g: AST[B, C],
  ) extends AST[A, C]

  case Par[A1, A2, B1, B2](
    f1: AST[A1, B1],
    f2: AST[A2, B2],
  ) extends AST[(A1, A2), (B1, B2)]

  case Dup[A]() extends AST[A, (A, A)]

  // ...
```

Demo Domain: **Workflows**

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

- scripted *activities*

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

- scripted *activities*
- (often) long running (hours, days)

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

- scripted *activities*
- (often) long running (hours, days)
- (often) mostly waiting

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

- scripted *activities*
- (often) long running (hours, days)
- (often) mostly waiting
  - for activity to complete

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

- scripted *activities*
- (often) long running (hours, days)
- (often) mostly waiting
  - for activity to complete
  - for human input

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

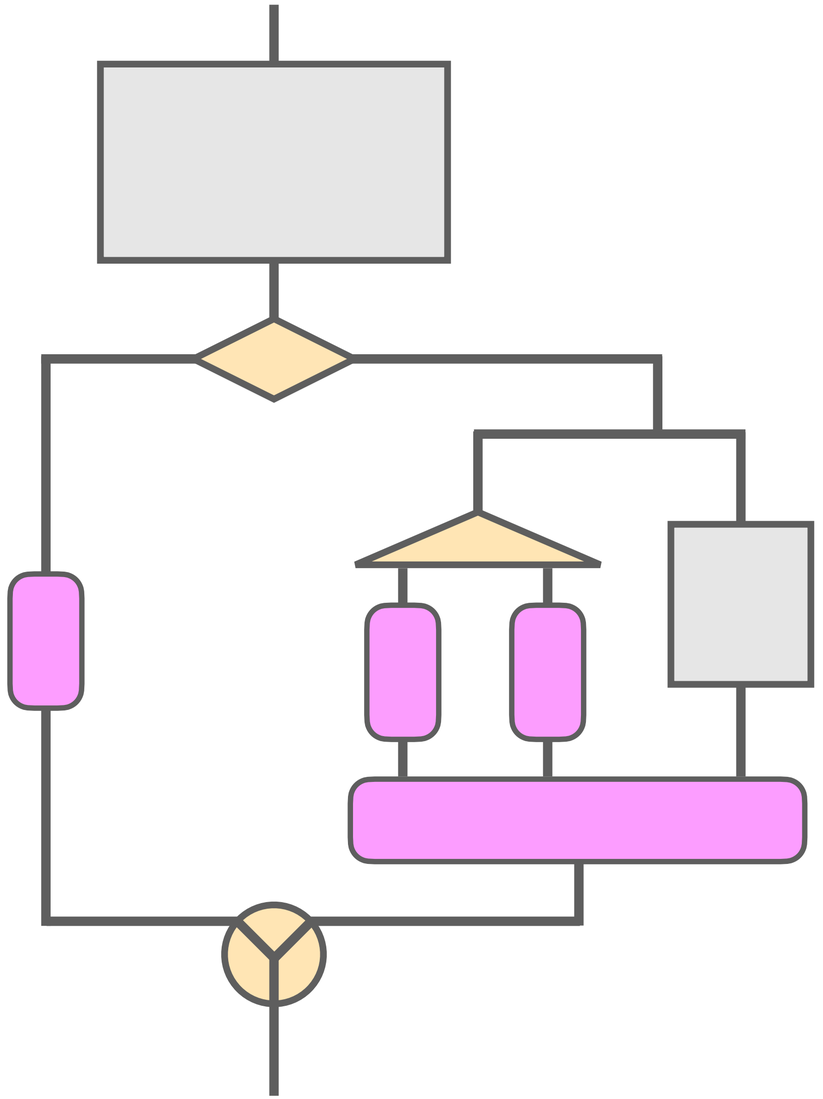
- scripted *activities*
- (often) long running (hours, days)
- (often) mostly waiting
  - for activity to complete
  - for human input
- require **durable execution**

Demo Domain: **Workflows**

“orchestrated and repeatable patterns of activity” — Wikipedia

- scripted *activities*
- (often) long running (hours, days)
- (often) mostly waiting
  - for activity to complete
  - for human input
- require **durable execution**
  - can’t assume to stay in memory for the whole execution

Workflow: Example



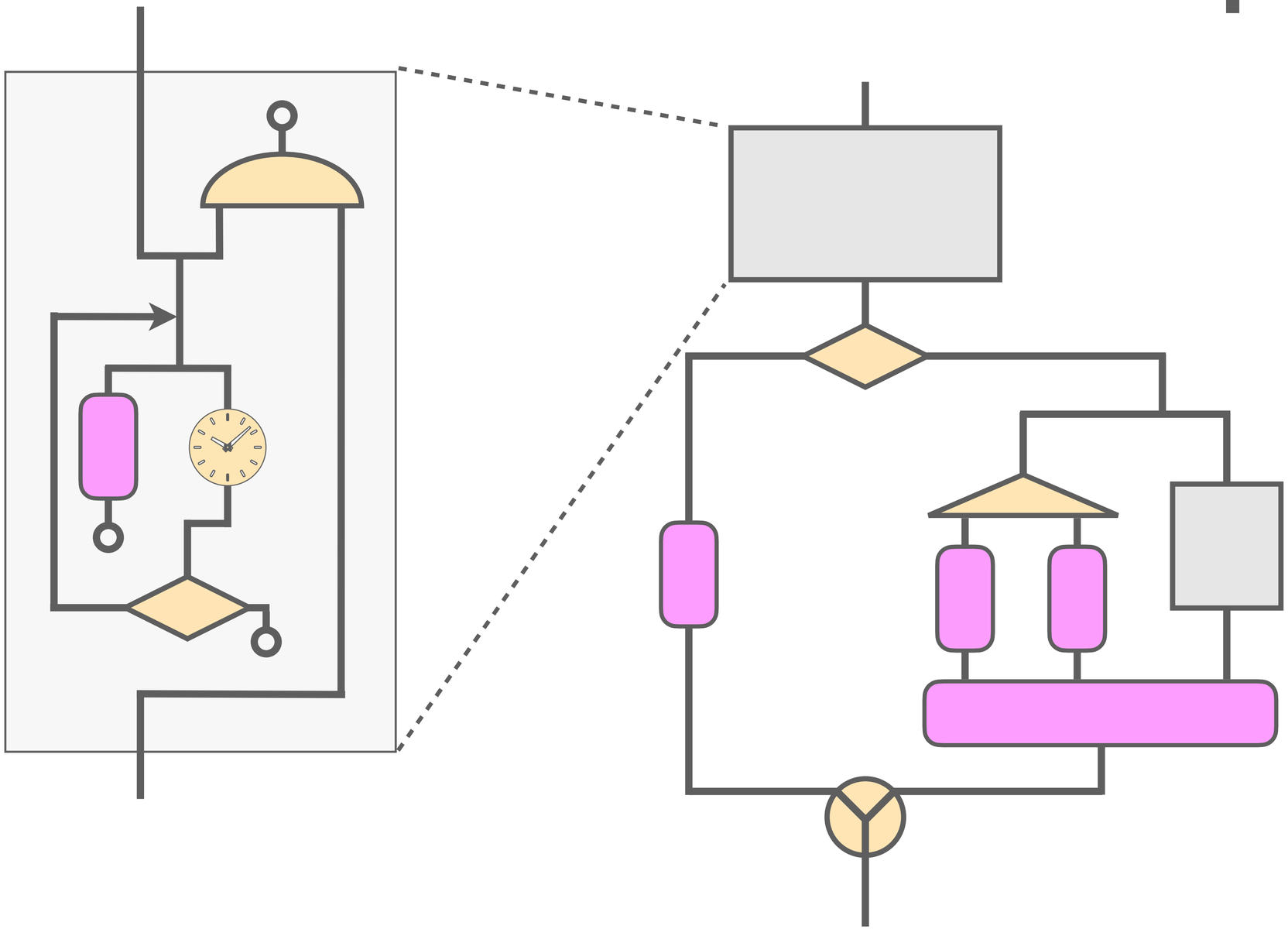
nested workflow

built-in operator

activity

unit-typed input/output

Workflow: Example



nested workflow

built-in operator

activity

unit-typed input/output

Workflow DSL Requirements

Workflow DSL Requirements

- expressive control flow

Workflow DSL Requirements

- expressive control flow
  - branching

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering
  - statically

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering
  - statically
  - mid-execution

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering
  - statically
  - mid-execution
- static analyses

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering
  - statically
  - mid-execution
- static analyses
- execution traces

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering
  - statically
  - mid-execution
- static analyses
- execution traces
- switch to an external DSL later

Workflow DSL Requirements

- expressive control flow
  - branching
  - loops
  - parallel processing
  - sharing intermediate results
- (reasonable) type-safety
  - inherited from the host lang
- **executable durably**

Maybe later

- graphical rendering
  - statically
  - mid-execution
- static analyses
- execution traces
- switch to an external DSL later
  - *without having to rewrite old workflows*

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history
- Concurrently

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history
- Concurrently
  - Check criminal record

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history
- Concurrently
  - Check criminal record
  - Check civil record

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history
- Concurrently
  - Check criminal record
  - Check civil record
  - Verify employment history

Demo Sub-Domain: **Background Check**

Inspired by https://learn.temporal.io/examples/go/background-checks/

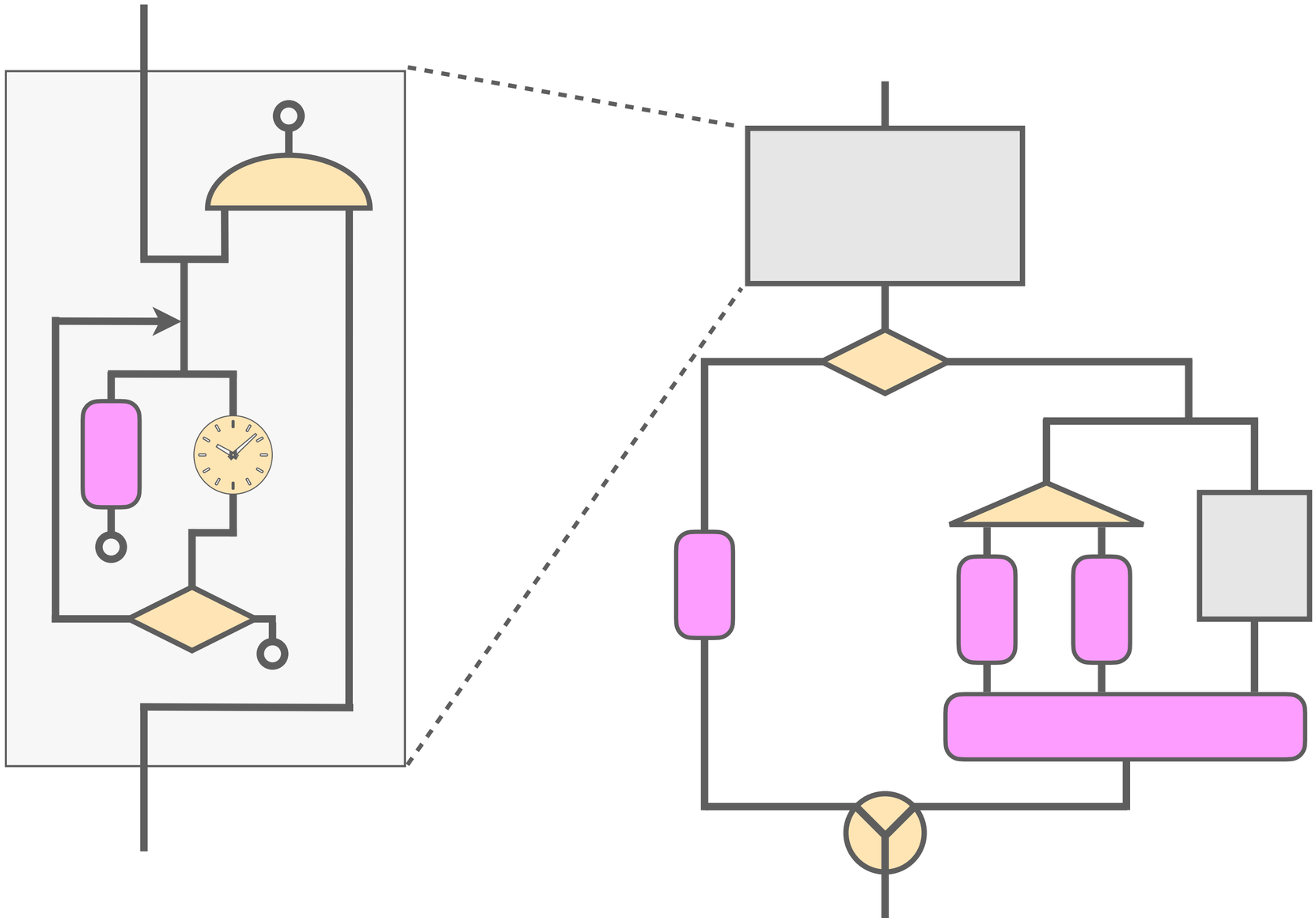
- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history
- Concurrently
  - Check criminal record
  - Check civil record
  - Verify employment history
    - Notify a human researcher

Demo Sub-Domain: **Background Check**

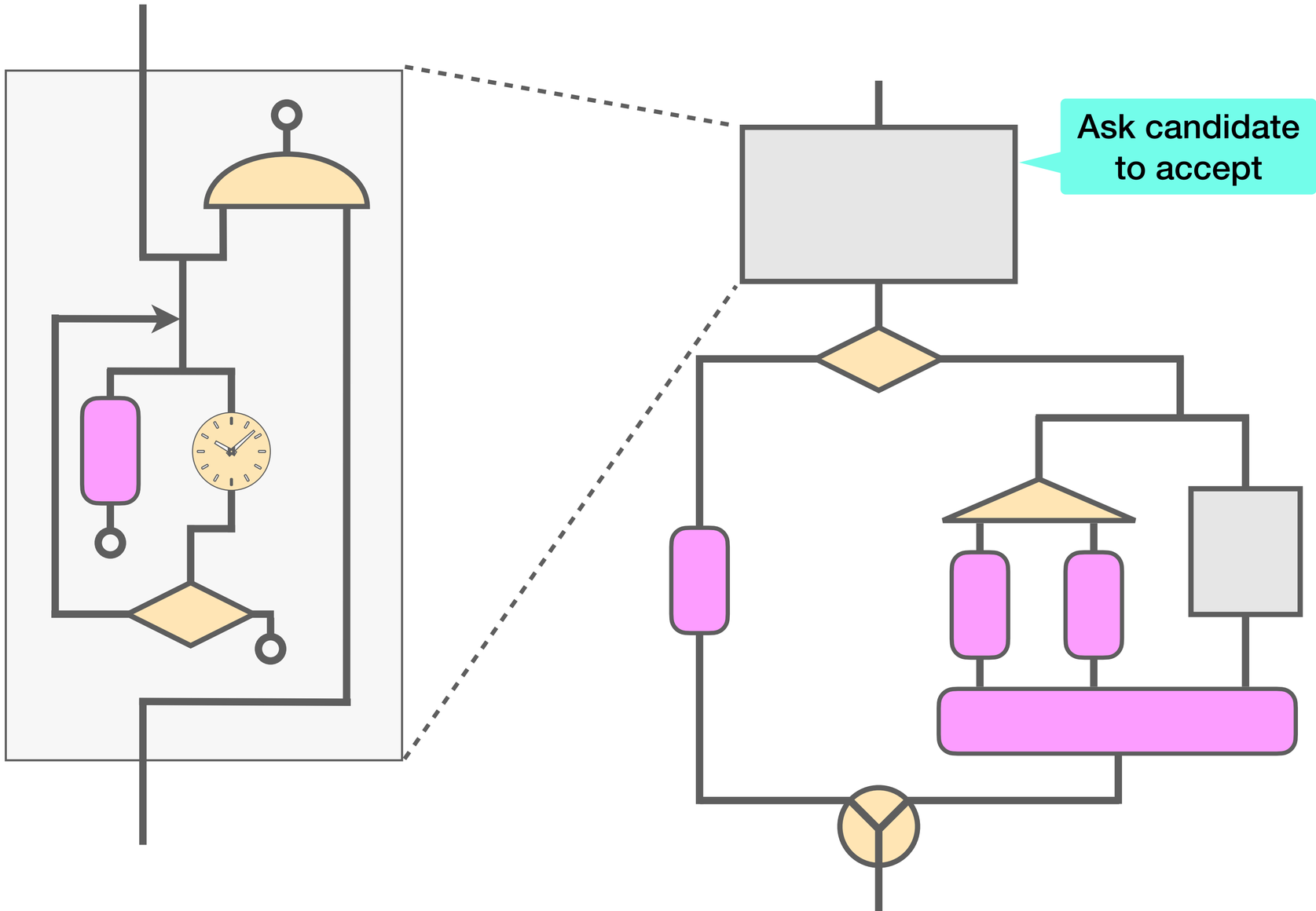
Inspired by https://learn.temporal.io/examples/go/background-checks/

- HR person initiates
  - Inputs candidate email
- Ask candidate (via email) to accept
- Candidate accepts by providing
  - Personal info
  - Employment history
- Concurrently
  - Check criminal record
  - Check civil record
  - Verify employment history
    - Notify a human researcher
- Produce report (for the HR person)

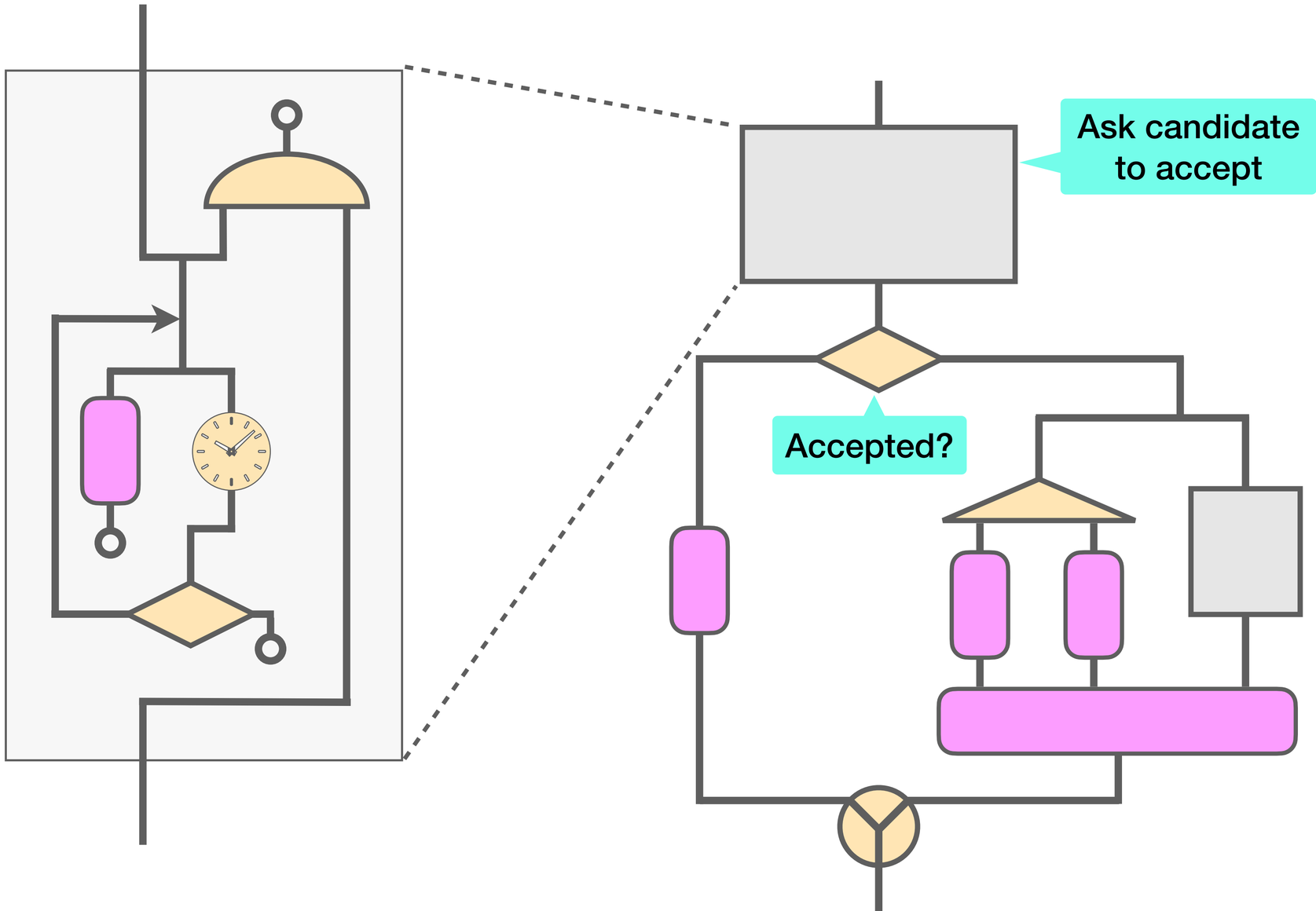
Background Check Workflow



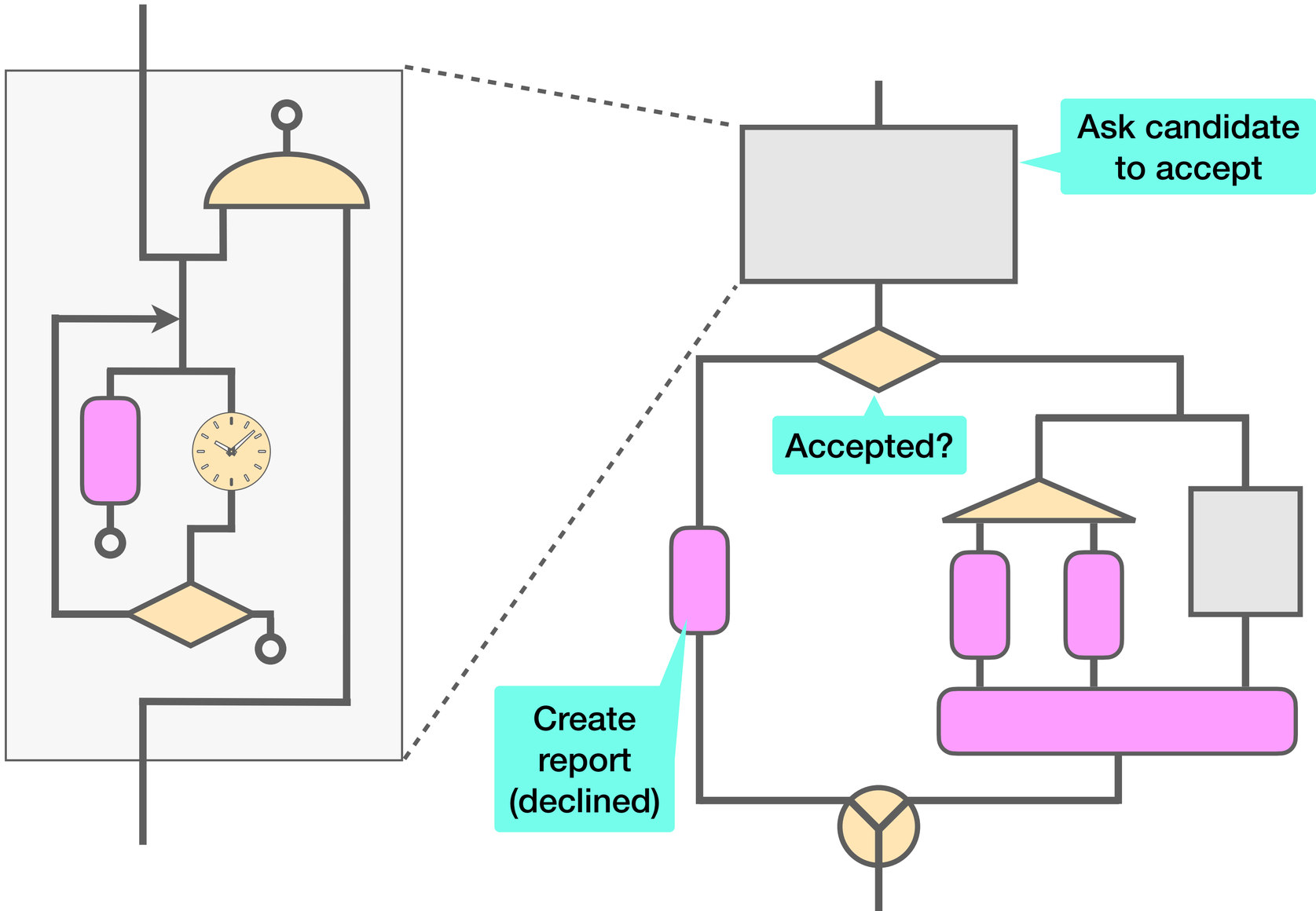
Background Check Workflow



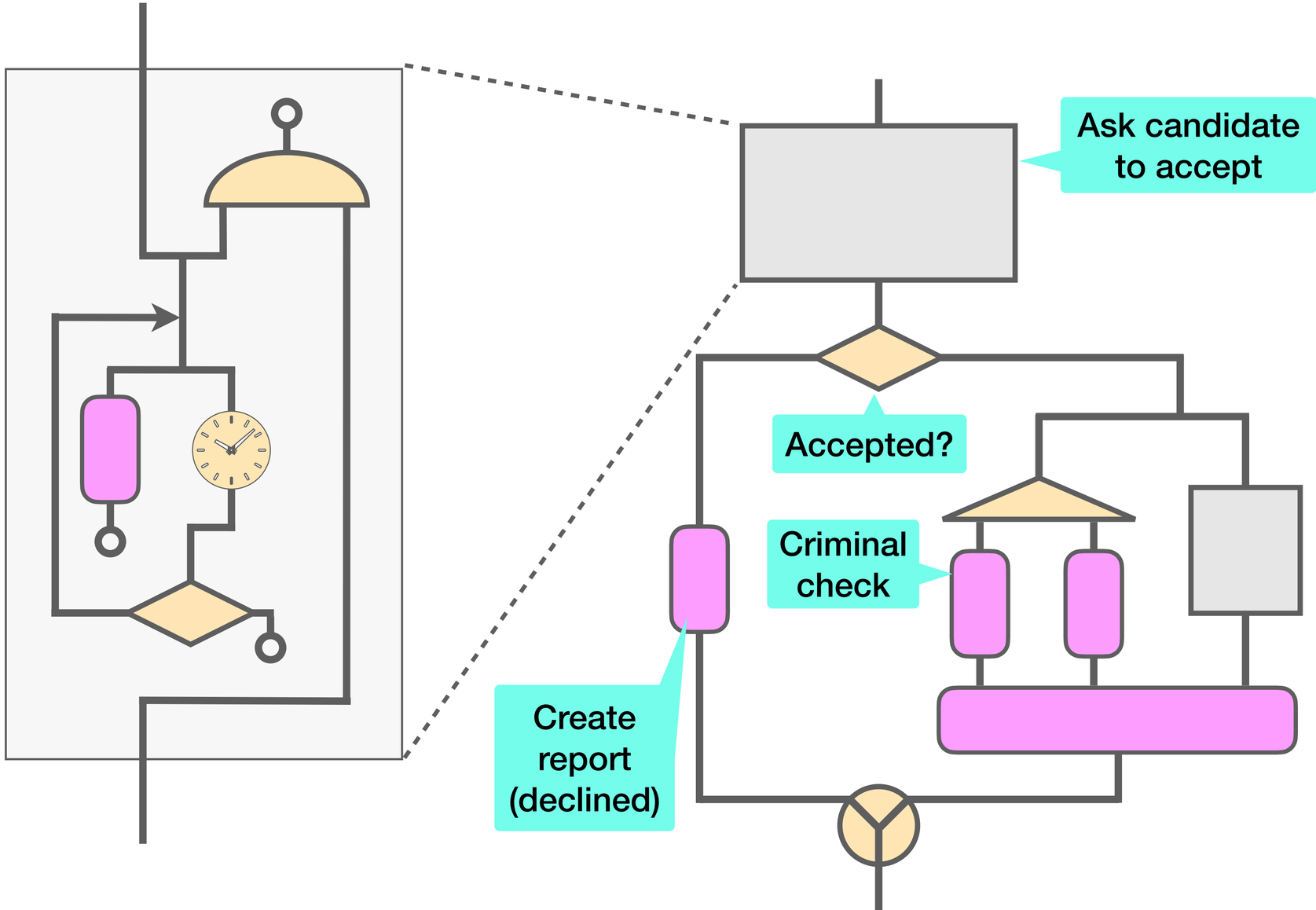
Background Check Workflow



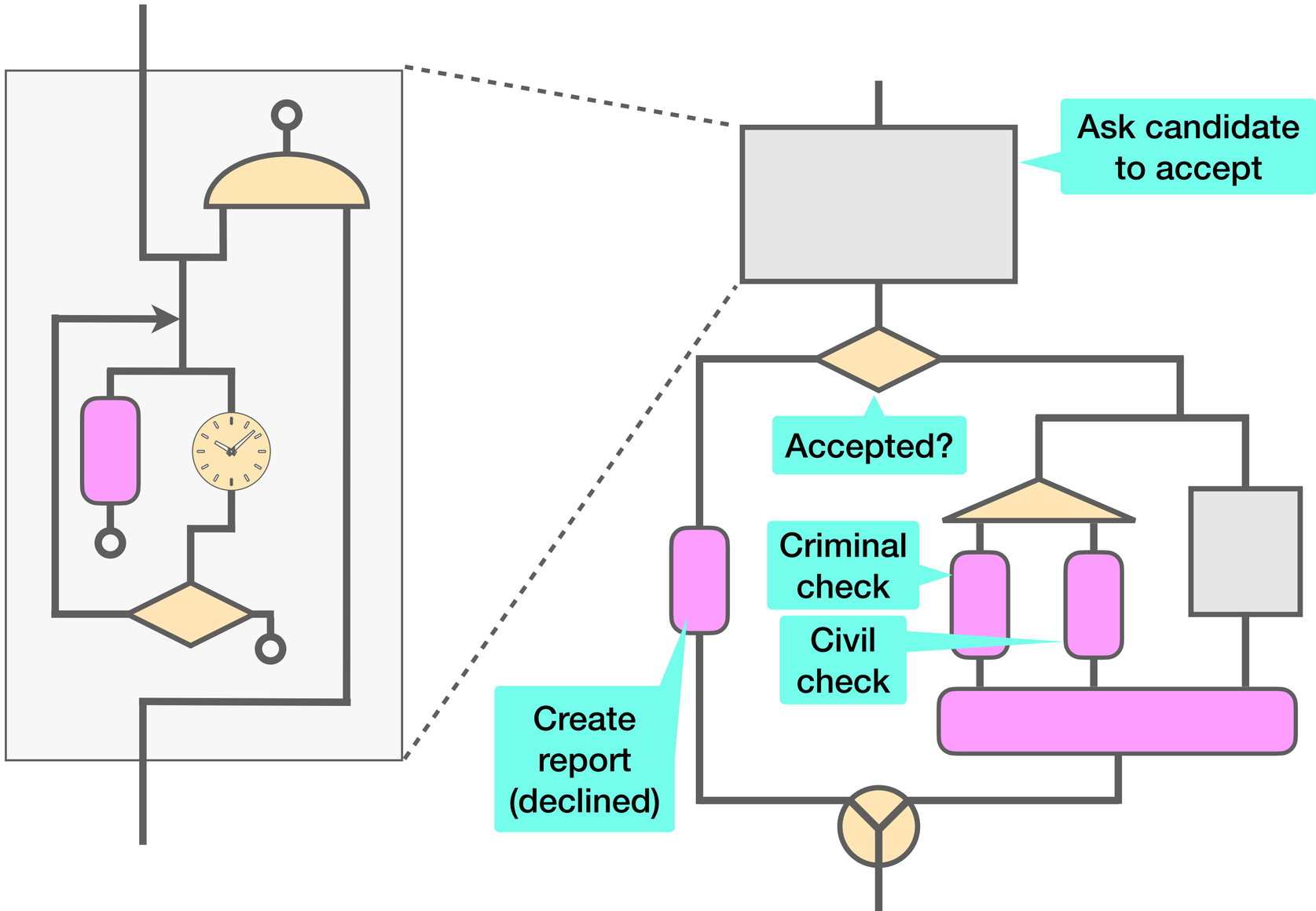
Background Check Workflow



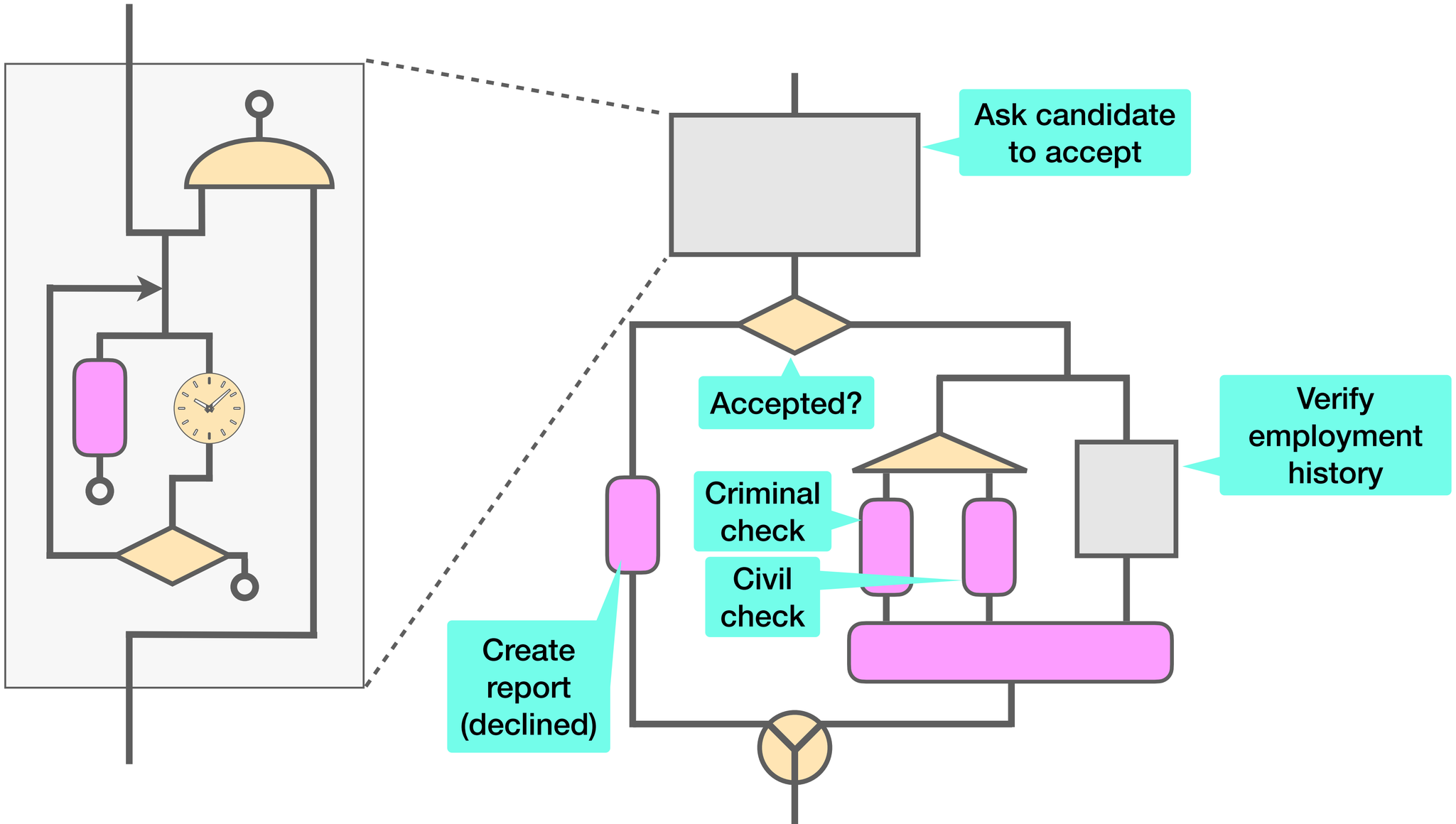
Background Check Workflow



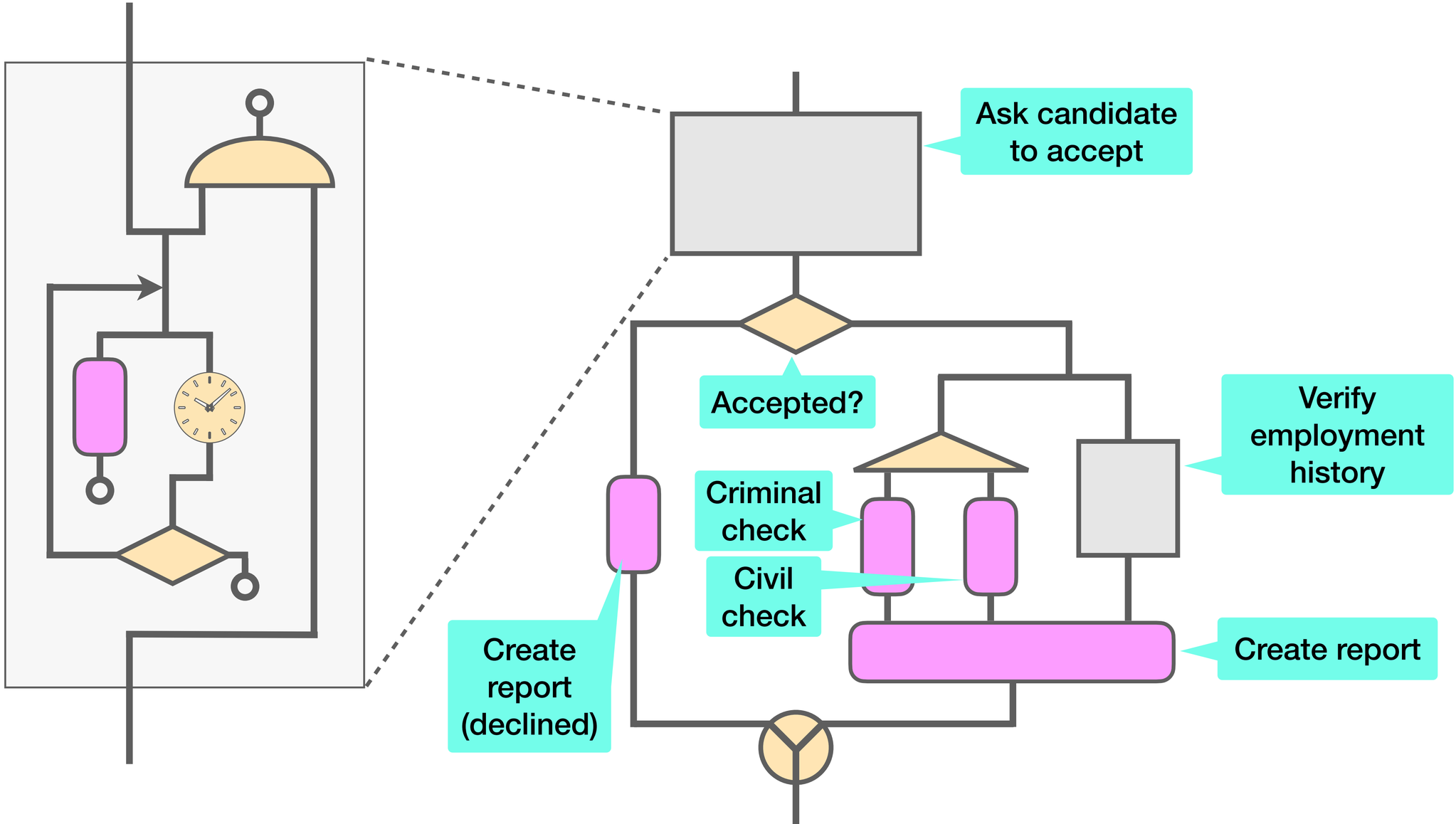
Background Check Workflow



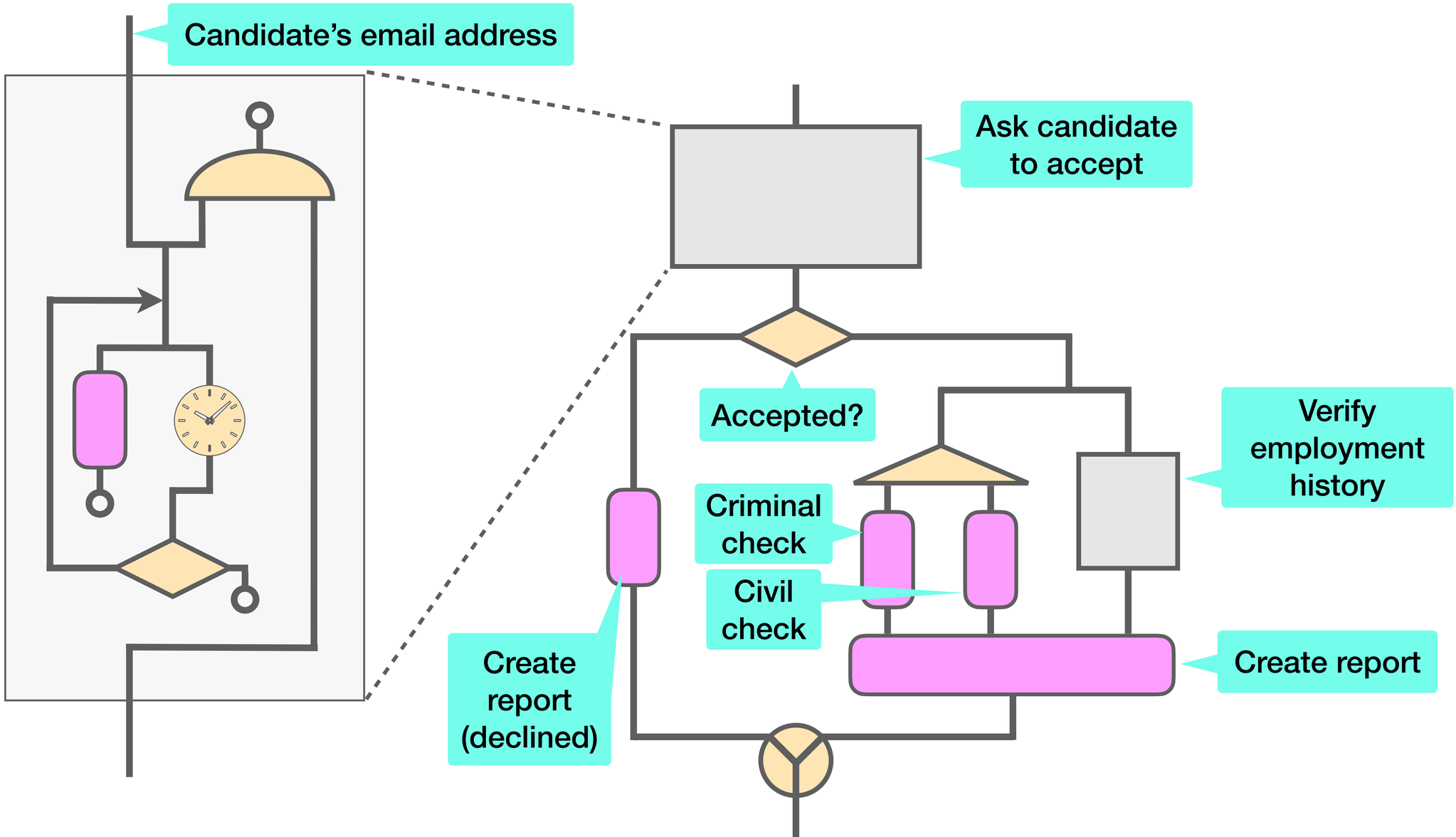
Background Check Workflow



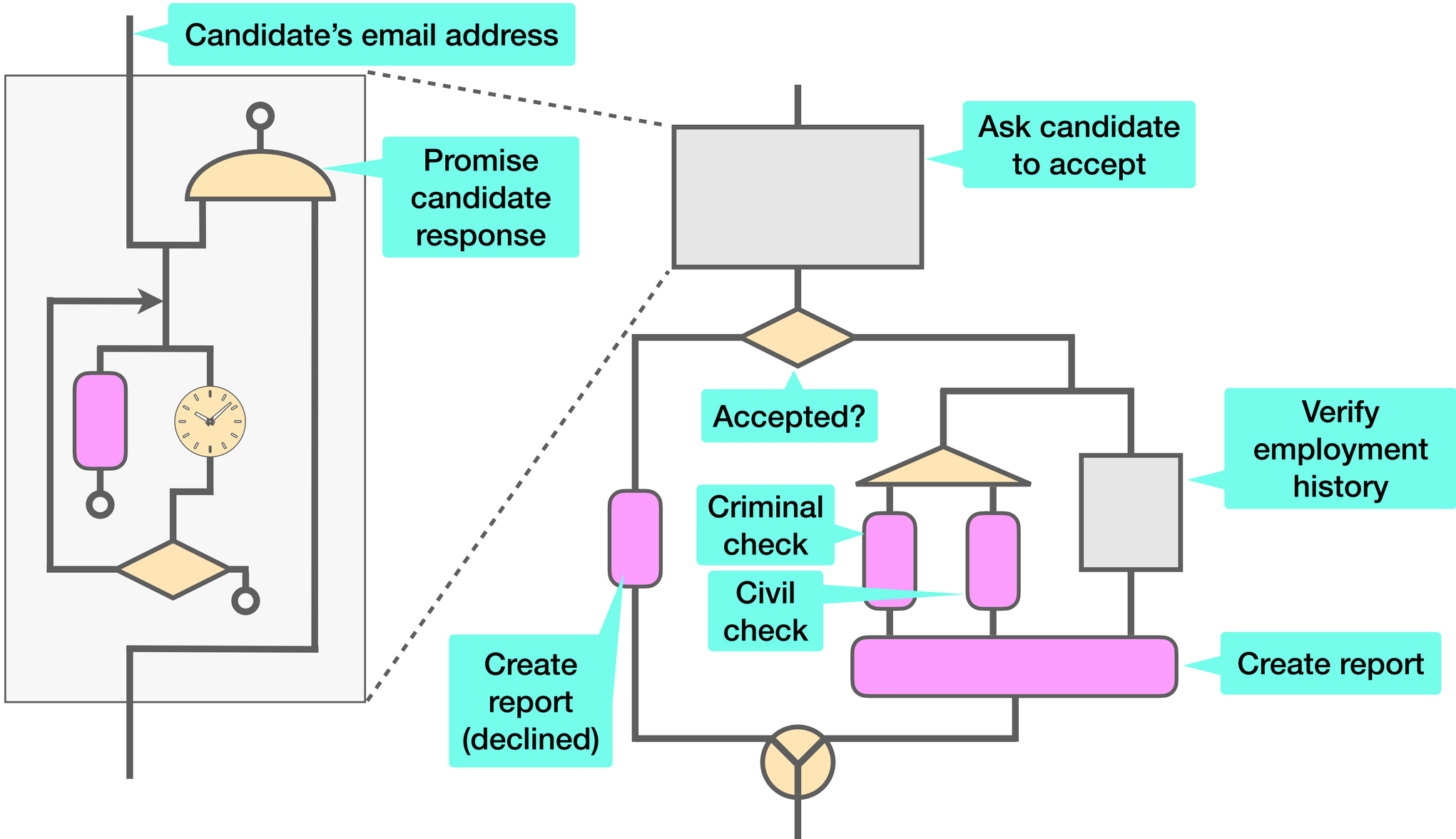
Background Check Workflow



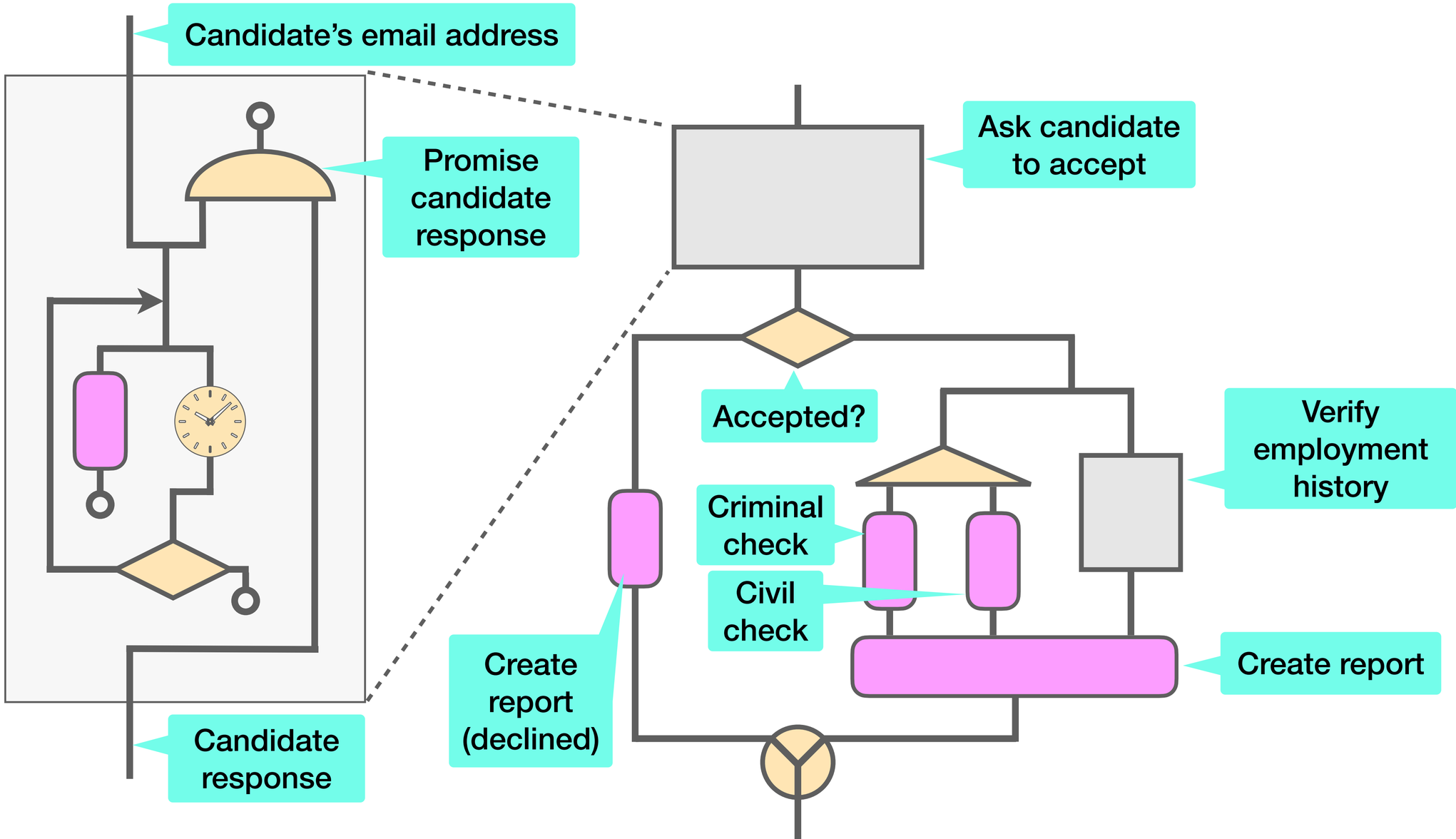
Background Check Workflow



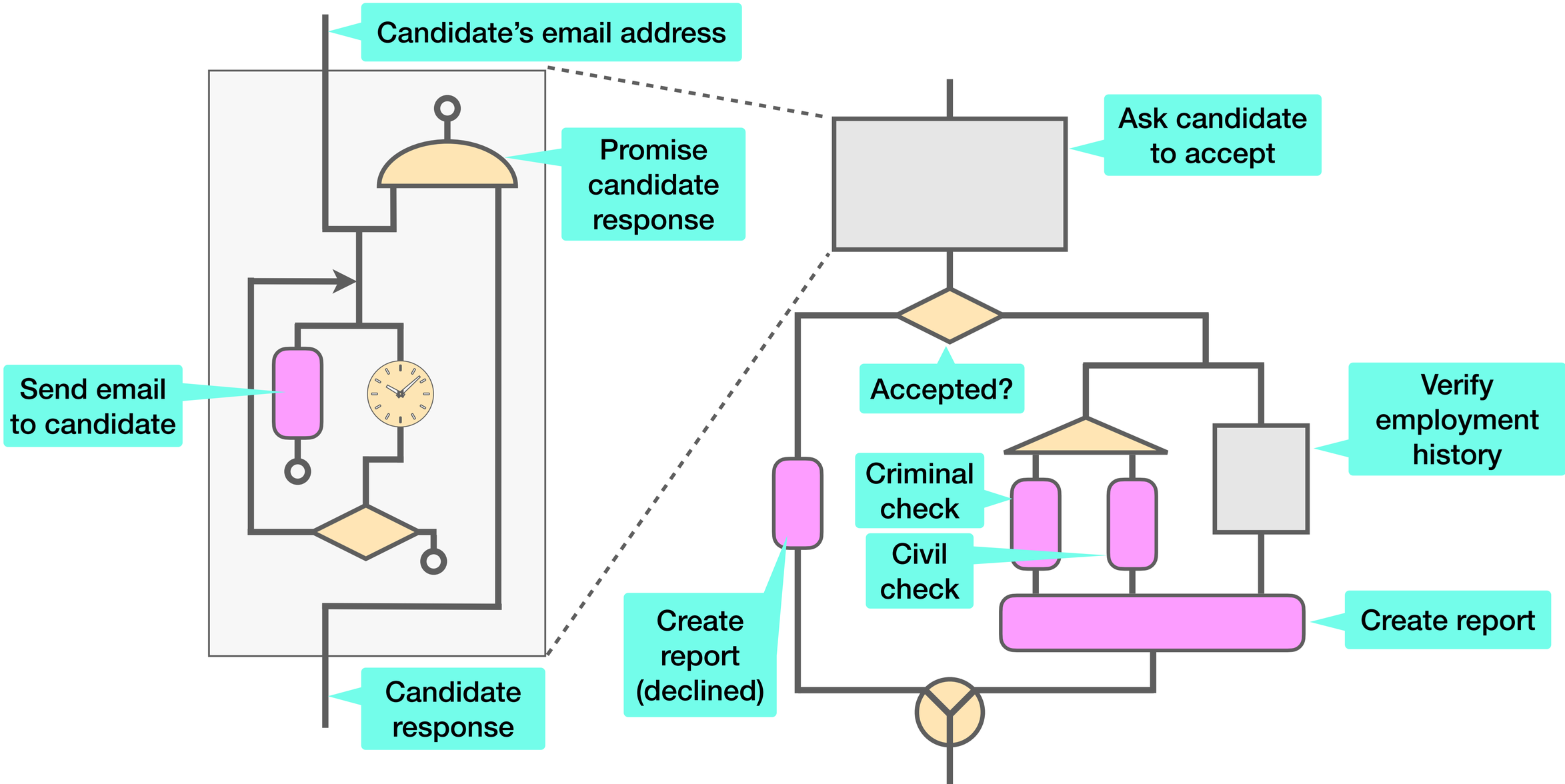
Background Check Workflow



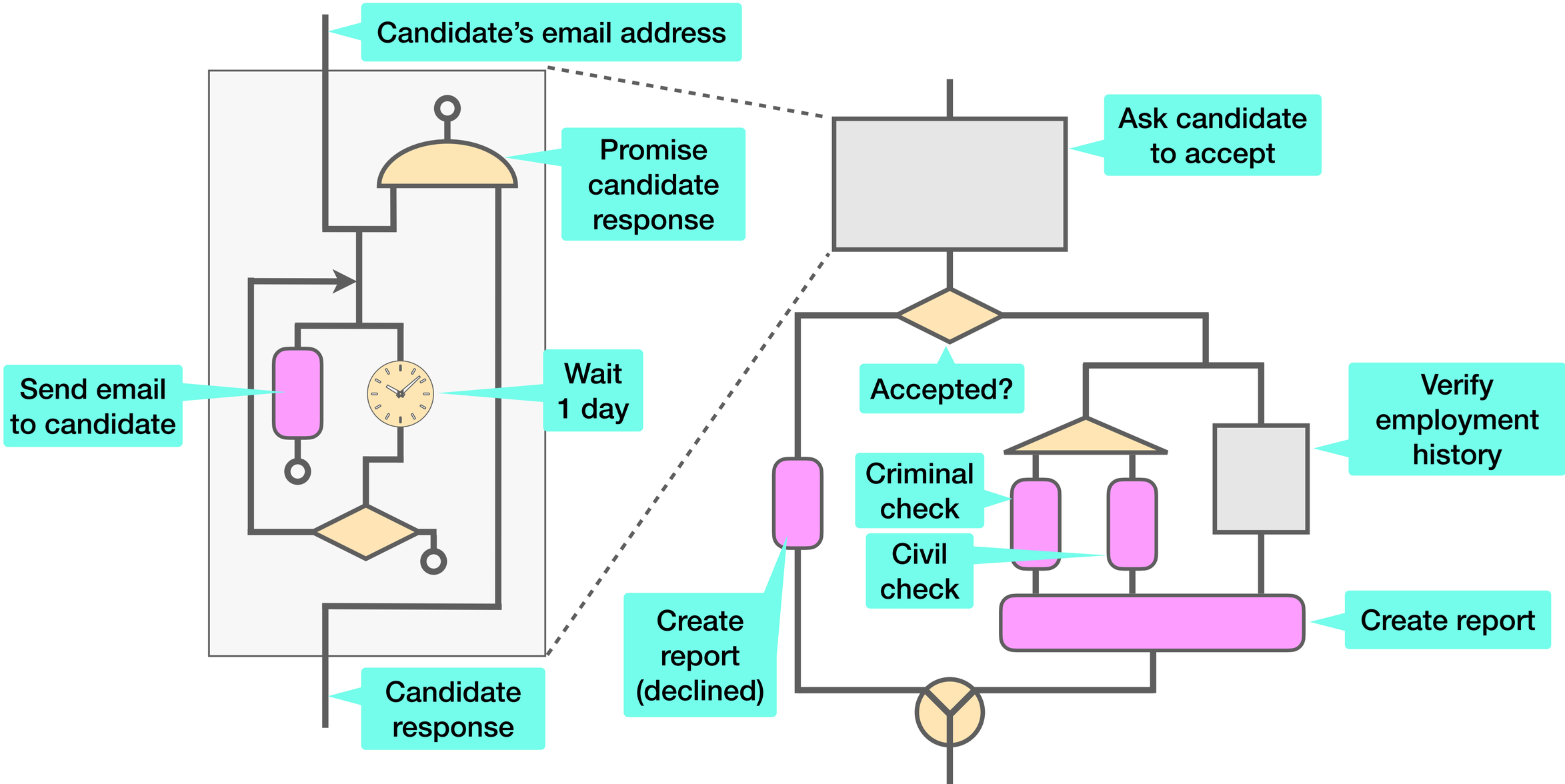
Background Check Workflow



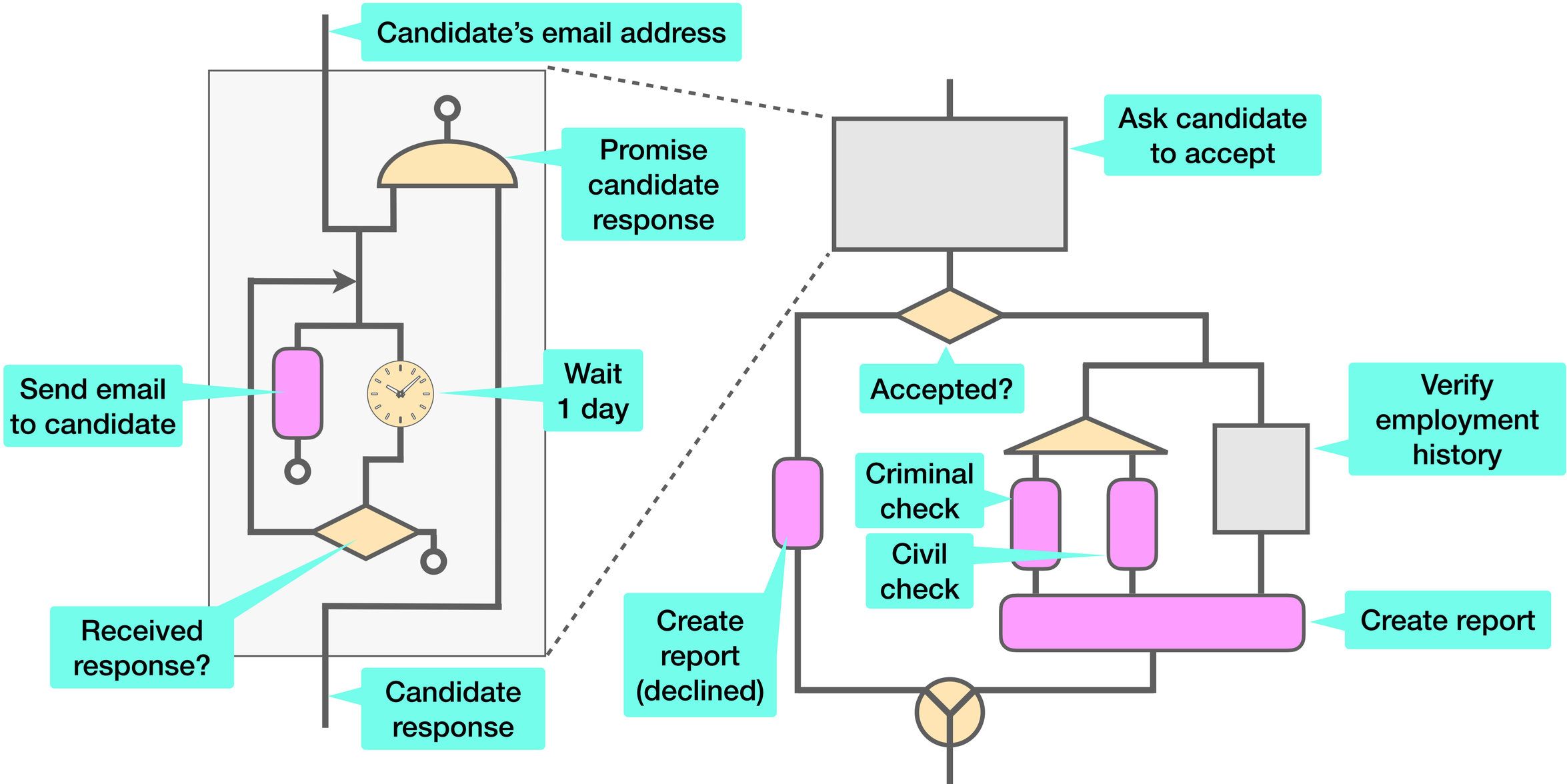
Background Check Workflow



Background Check Workflow



Background Check Workflow



User Code

```
val backgroundCheck: Flow[EmailAddress, Report] =
  Flow { candidate =>
    askForAcceptance(candidate) switch {
      case Left(x) =>
        Report.declined(x)
      case Right(personalId ** employmentHistory) =>
        val crimi = checkCrimi(personalId)
        val civil = checkCivil(personalId)
        val verif = verify(employmentHistory)
        Report.results(crimi ** civil ** verif)
    }
  }
```

👨‍💻 User Code

```
val backgroundCheck: Flow[EmailAddress, Report] =
  Flow { candidate =>
    askForAcceptance(candidate) switch {
      case Left(x) =>
        Report.declined(x)
      case Right(personalId ** employmentHistory) =>
        val crimi = checkCrimi(personalId)
        val civil = checkCivil(personalId)
        val verif = verify(employmentHistory)
        Report.results(crimi ** civil ** verif)
    }
  }
```

- Lambdas

👨‍💻 User Code

```
val backgroundCheck: Flow[EmailAddress, Report] =
  Flow { candidate =>
    askForAcceptance(candidate) switch {
      case Left(x) =>
        Report.declined(x)
      case Right(personalId ** employmentHistory) =>
        val crimi = checkCrimi(personalId)
        val civil = checkCivil(personalId)
        val verif = verify(employmentHistory)
        Report.results(crimi ** civil ** verif)
    }
  }
```

- Lambdas
- Pattern matching

User Code

```
val backgroundCheck: Flow[EmailAddress, Report] =
  Flow { candidate =>
    askForAcceptance(candidate) switch {
      case Left(x) =>
        Report.declined(x)
      case Right(personalId ** employmentHistory) =>
        val crimi = checkCrimi(personalId)
        val civil = checkCivil(personalId)
        val verif = verify(employmentHistory)
        Report.results(crimi ** civil ** verif)
    }
  }
```

- Lambdas
- Pattern matching
- Auxiliary variables

👨‍💻 User Code

```
val backgroundCheck: Flow[EmailAddress, Report] =
  Flow { candidate =>
    askForAcceptance(candidate) switch {
      case Left(x) =>
        Report.declined(x)
      case Right(personalId ** employmentHistory) =>
        val crimi = checkCrimi(personalId)
        val civil = checkCivil(personalId)
        val verif = verify(employmentHistory)
        Report.results(crimi ** civil ** verif)
    }
  }
```

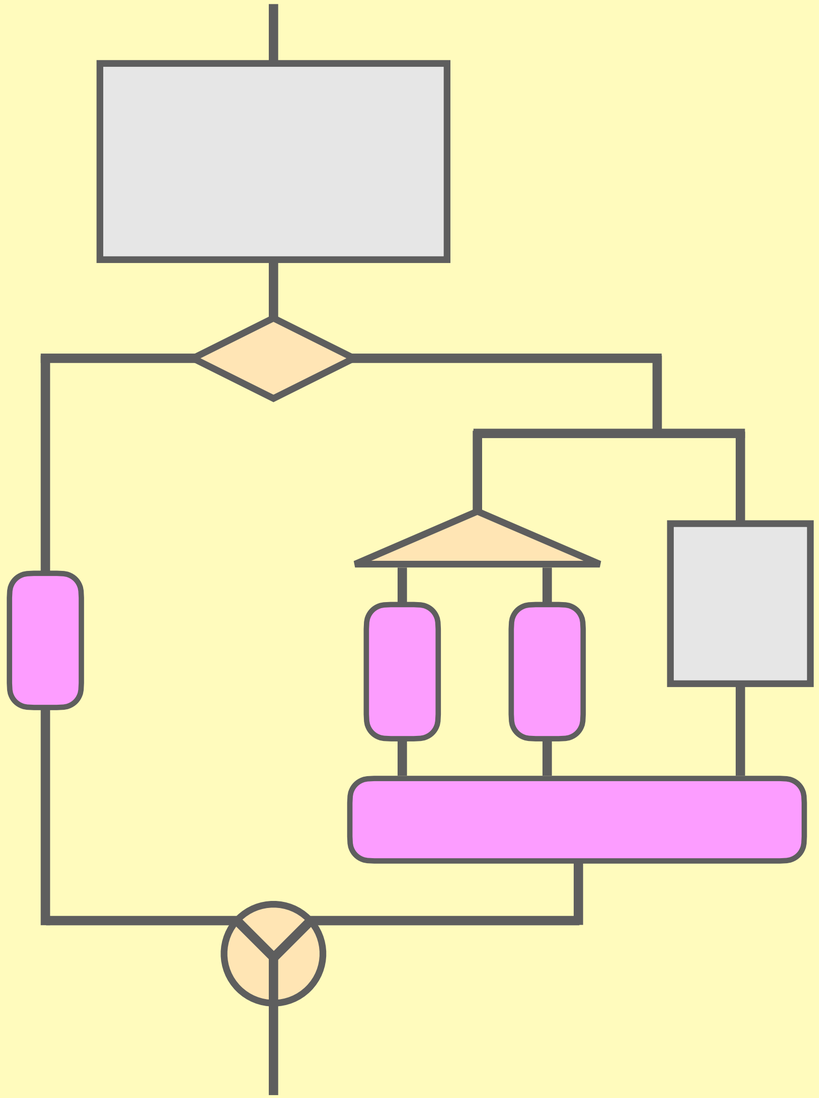
- Lambdas
- Pattern matching
- Auxiliary variables

Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```

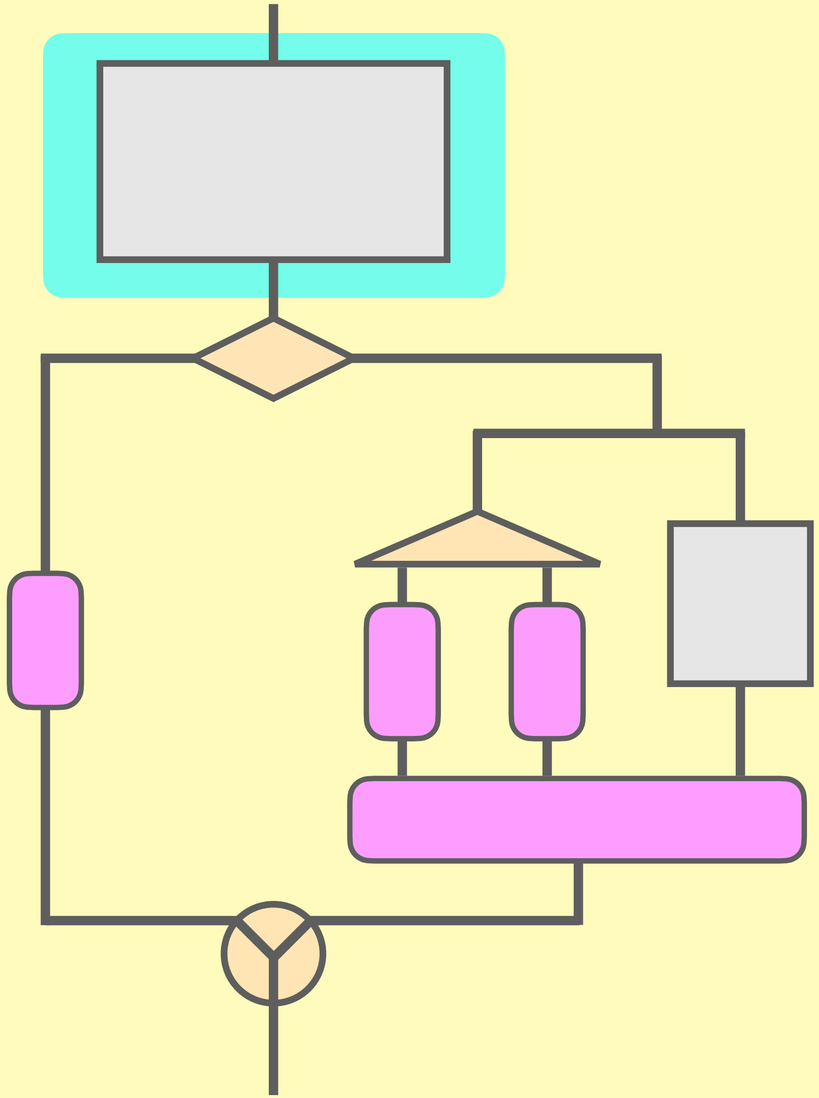
👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



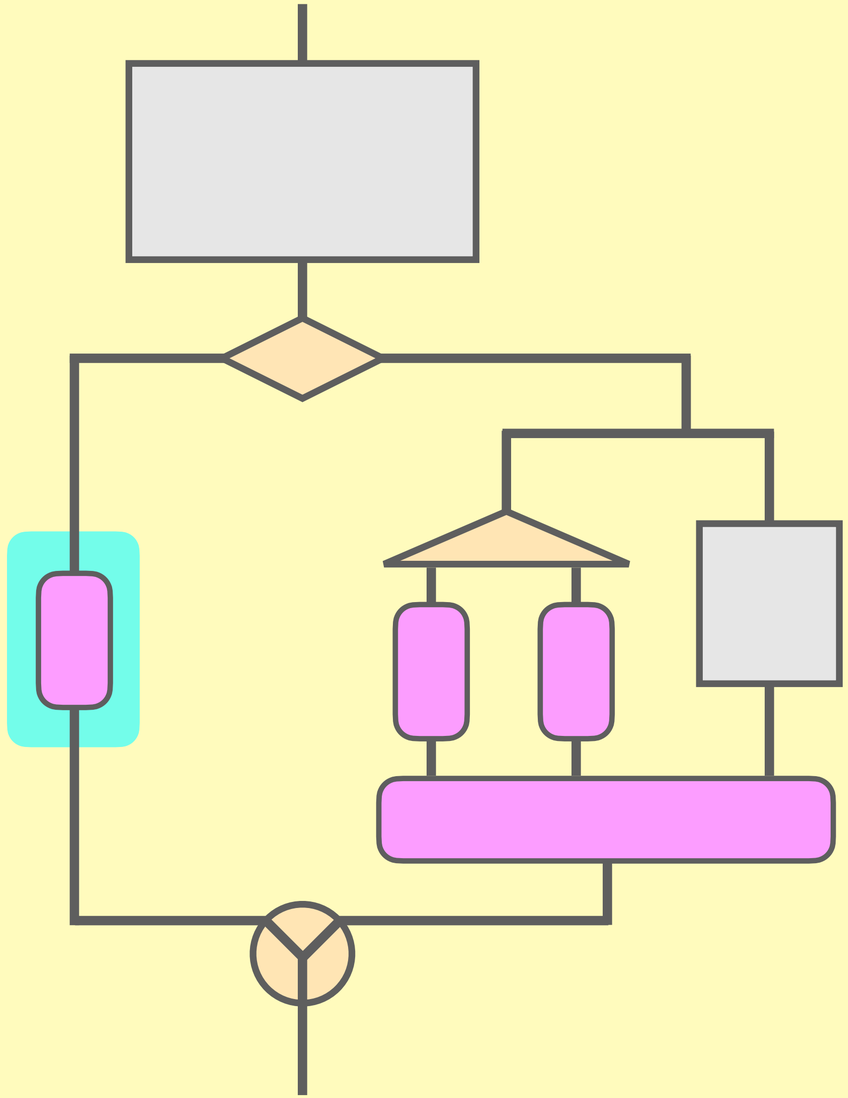
👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



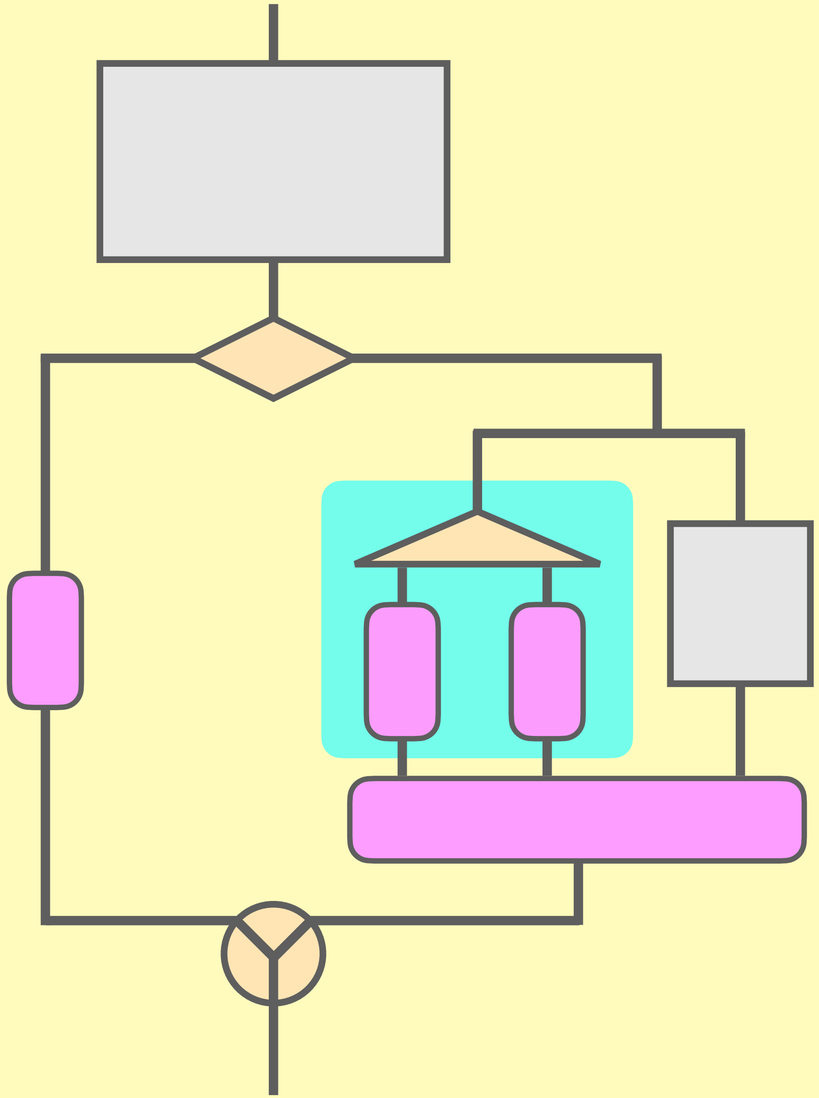
👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



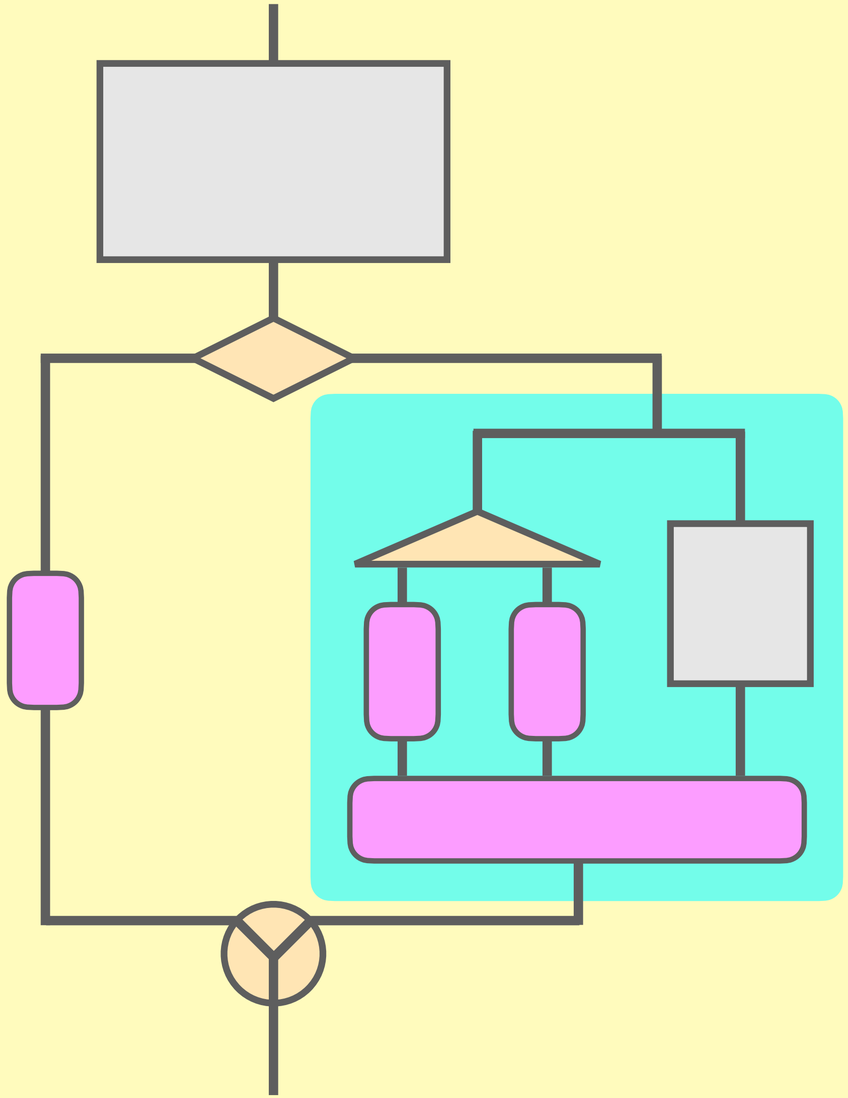
👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



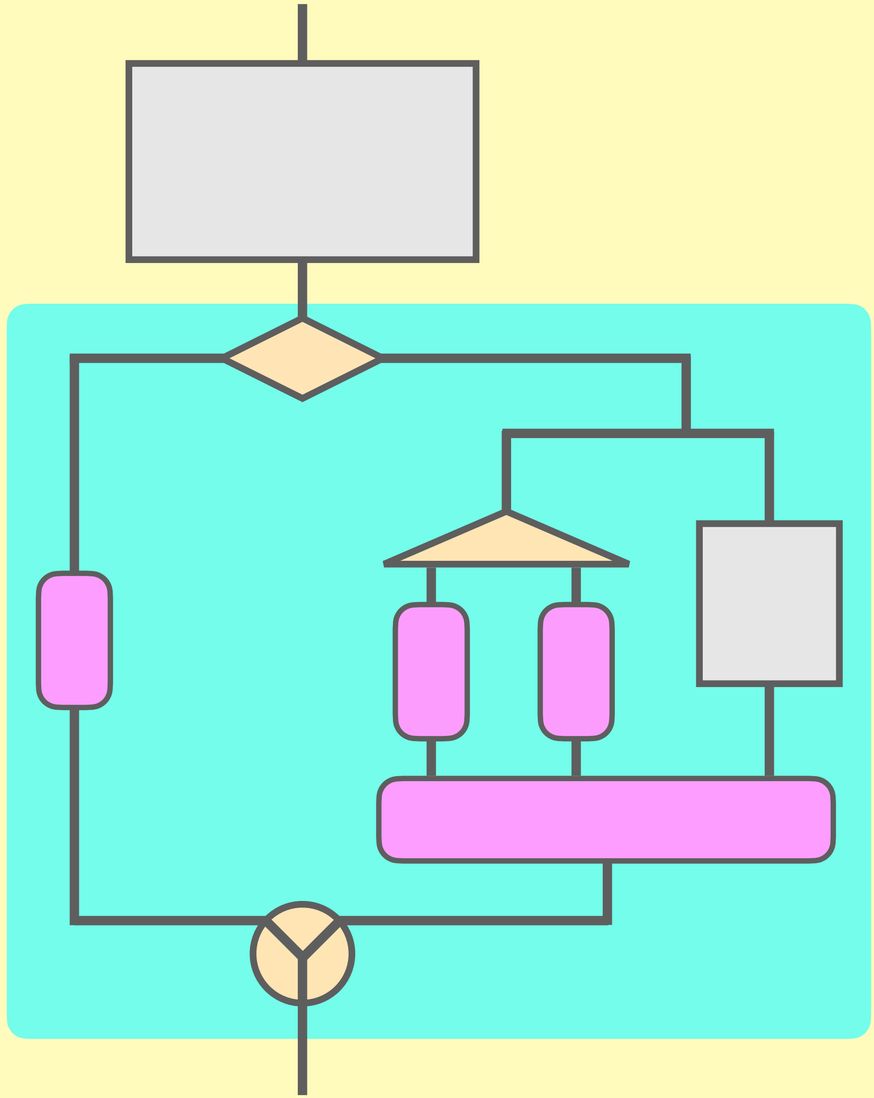
👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



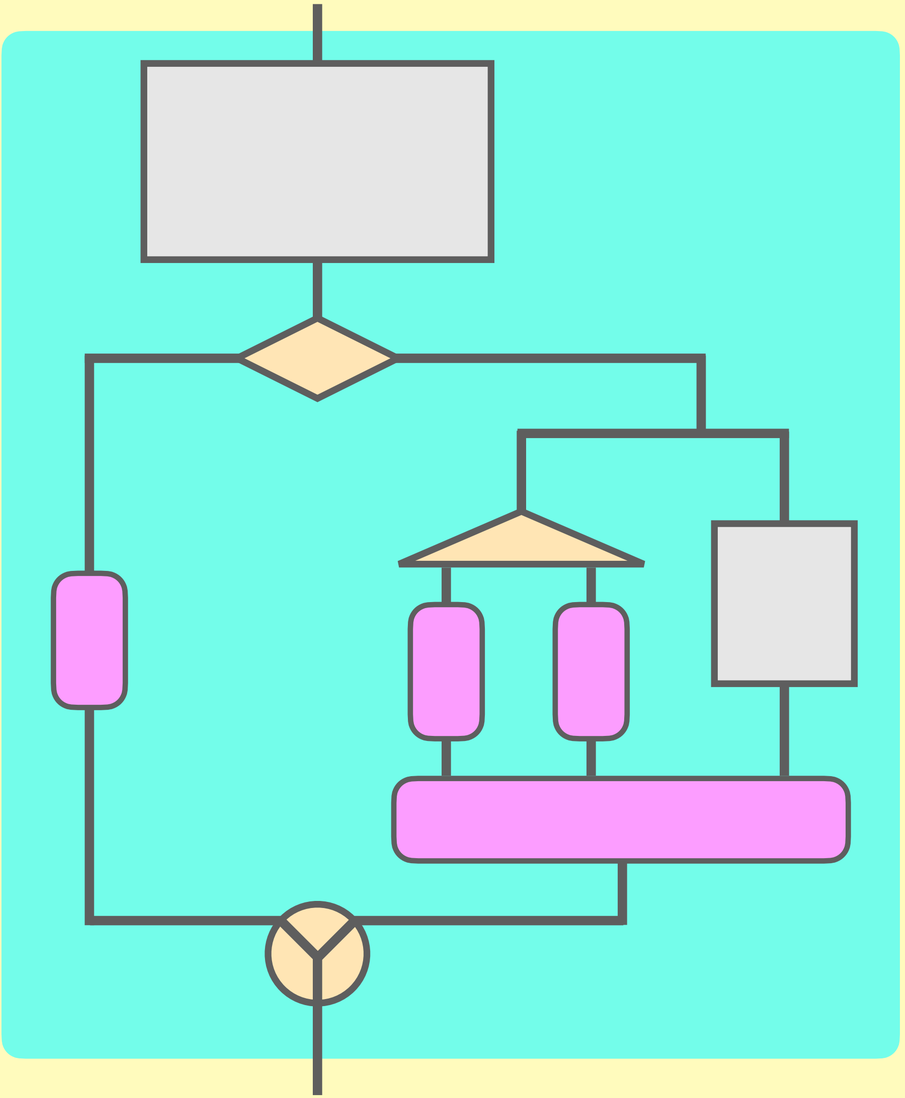
Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



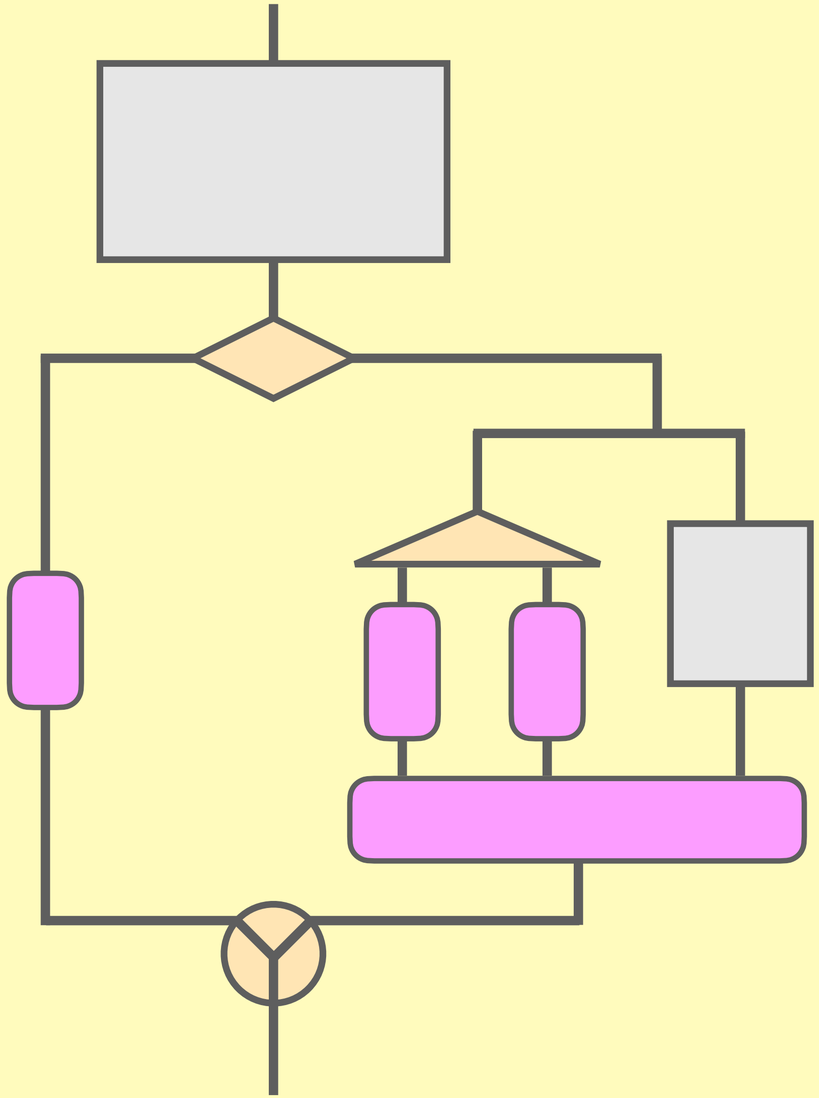
👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



👩‍🔬 Internal Representation

```
AndThen(
  askForAcceptance,
  Switch(
    Report.declined,
    AndThen(
      Par(
        AndThen(
          Dup(),
          Par(checkCrimi, checkCivil)
        ),
        verify
      ),
      Report.results
)))
```



Goal

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

Goal

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

Goal

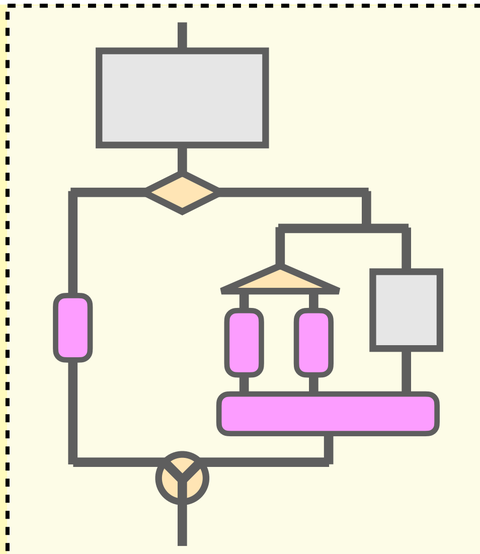
```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```


```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```

Goal

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

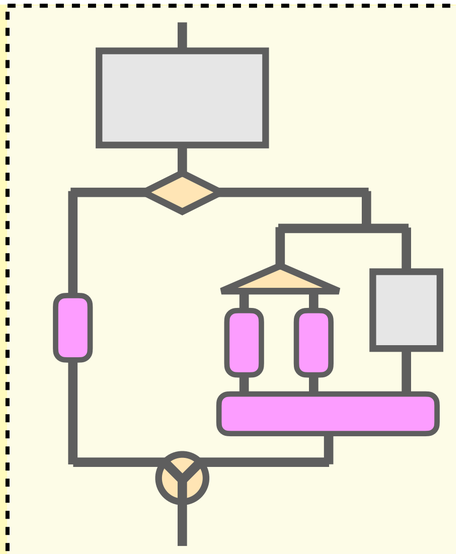
```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



Goal

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

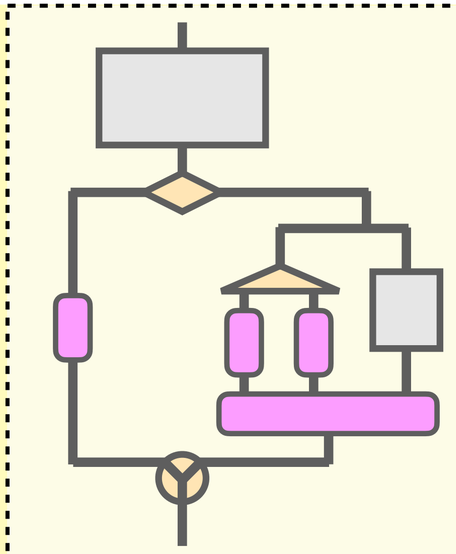

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



Goal

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```


```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



```
case Right(id ** history) =>
 val crimi = checkCrimi(id)
 val civil = checkCivil(id)
 val verif = verify(history)
 results(crimi ** civil ** verif)
```

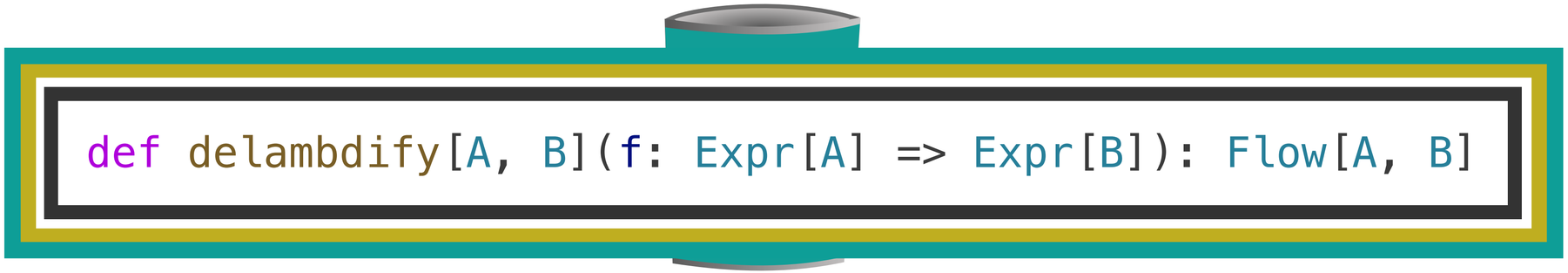
```
case       (id ** history) =>
 val crimi = checkCrimi(id)
 val civil = checkCivil(id)
 val verif = verify(history)
 results(crimi ** civil ** verif)
```

```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

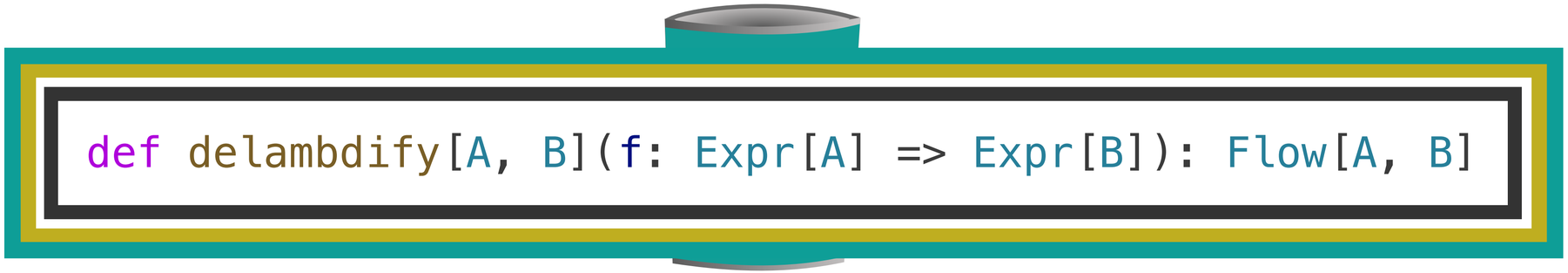

```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```




```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```




```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

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

```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable()
```

```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable()
  val expr : Expr[B] = f(🍅)
```

```
val onAccept:  Expr[PersonalId ** EmploymentHistory]  =>  Expr[Report]  =
  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]

expr : Expr[Report] =
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case      (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

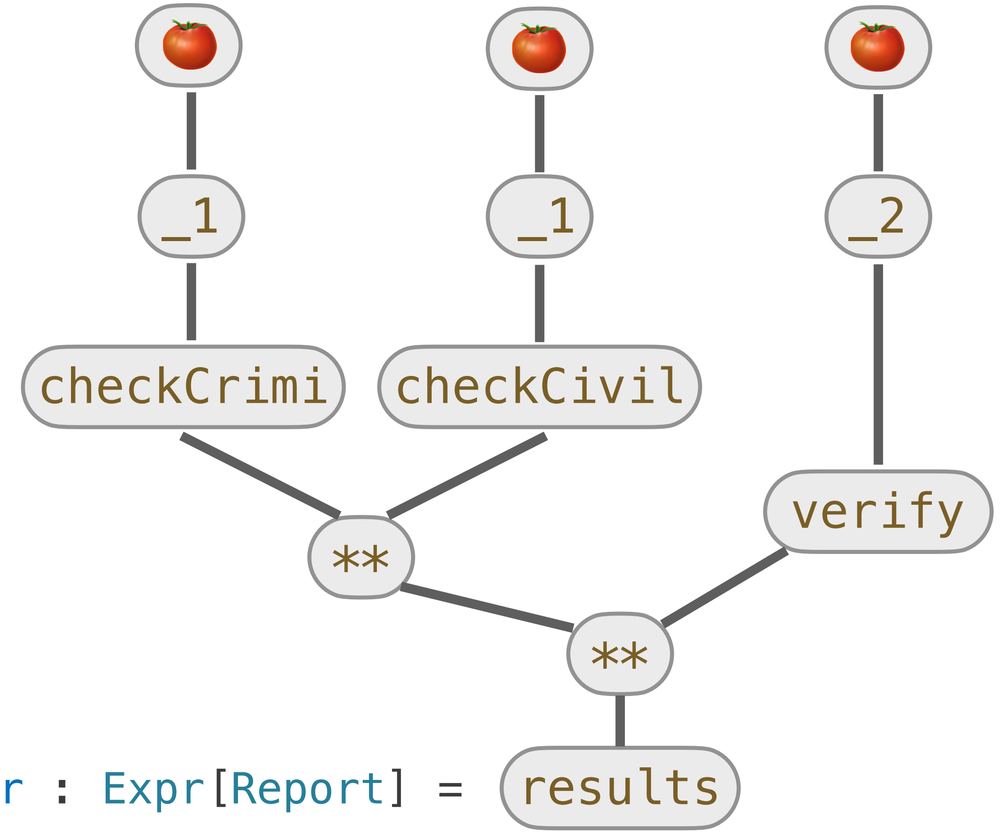
```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case       (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```




```
expr : Expr[Report] =
```

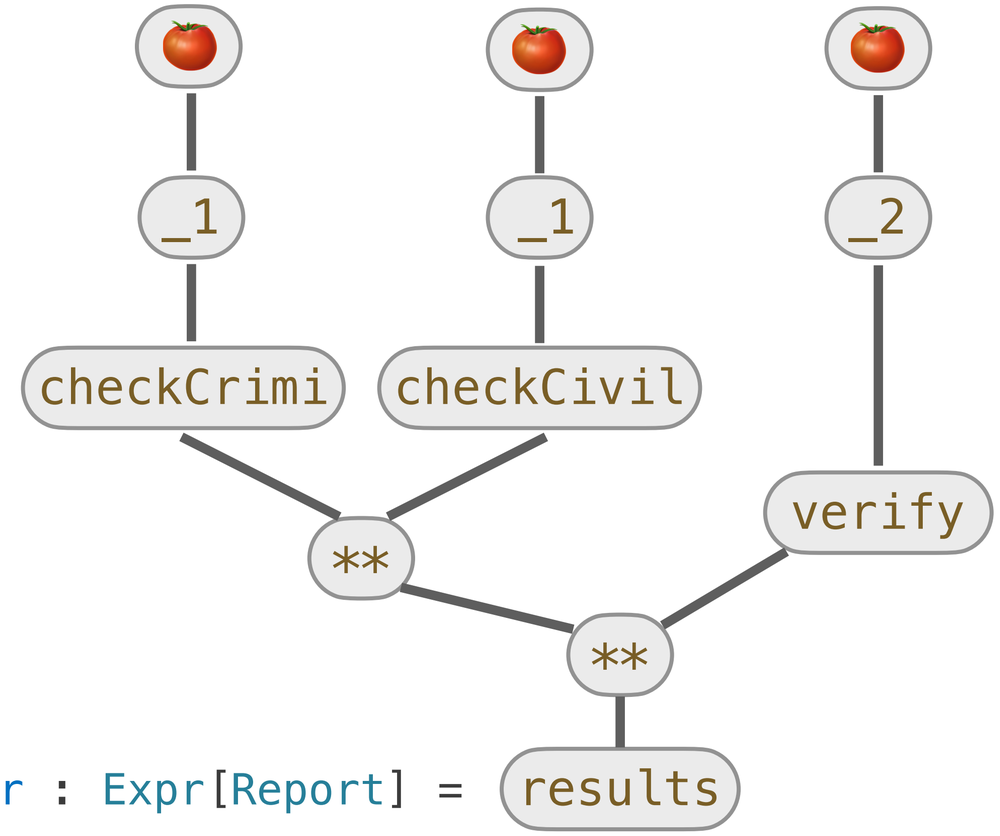
```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  Lam(🍅, expr)
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case       (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



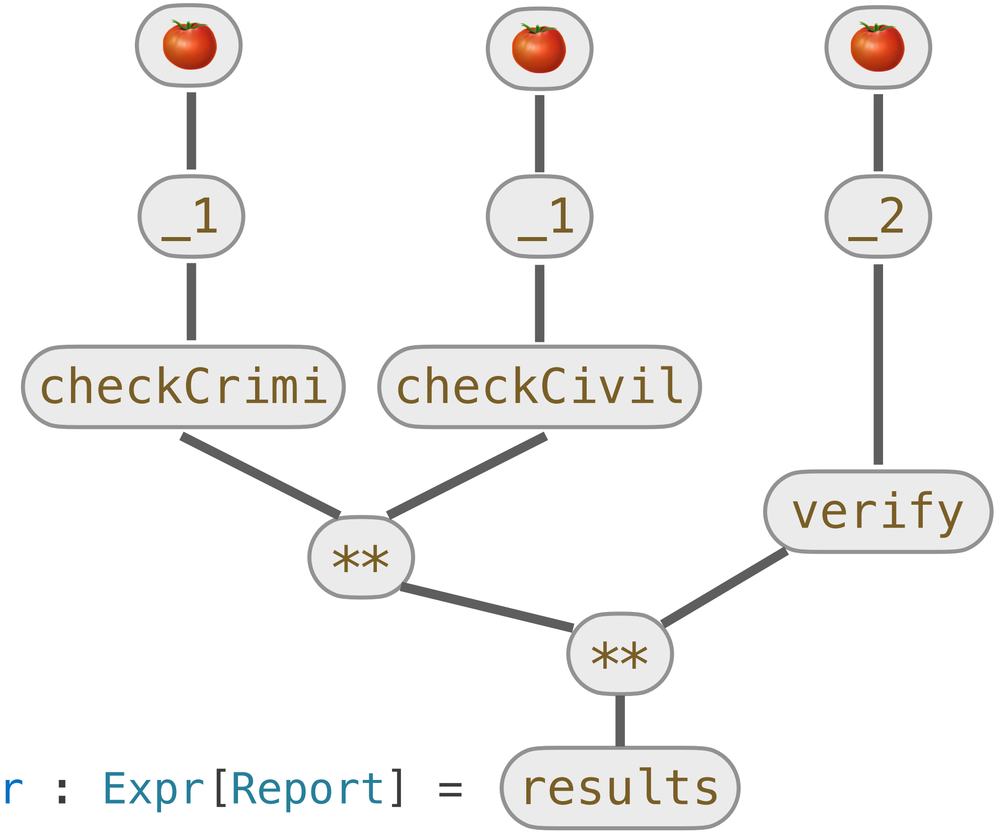
```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  Lam(🍅, expr)
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case       (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```




```
expr : Expr[Report] =
```

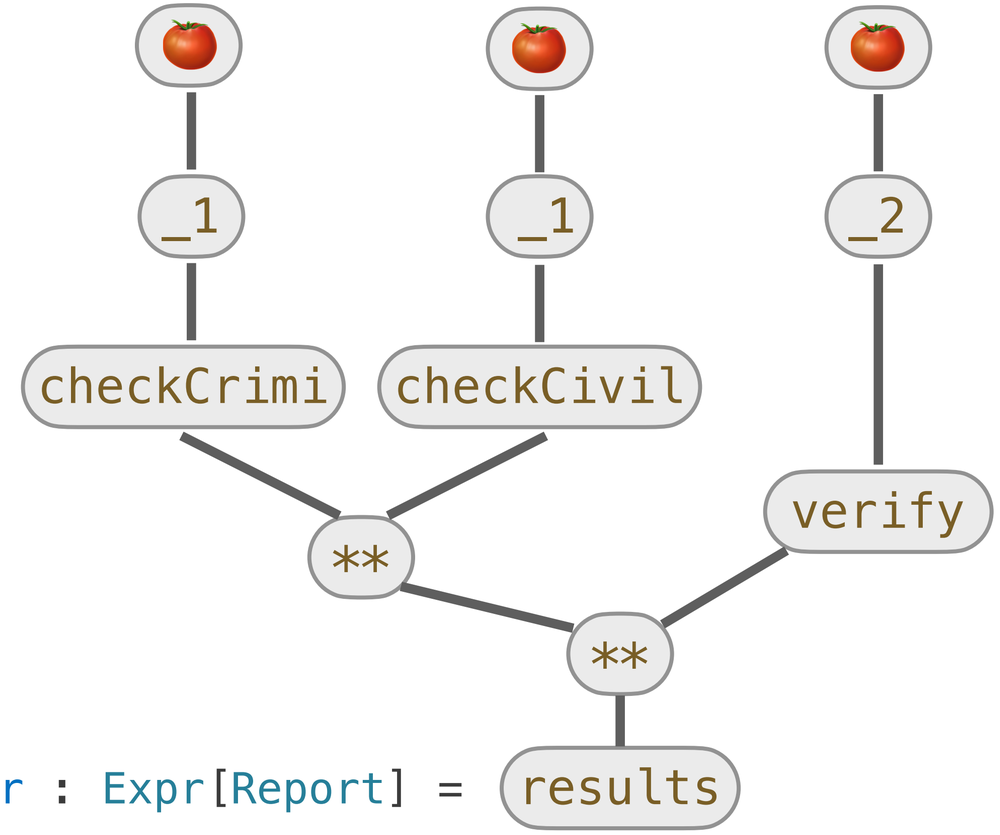
```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case       (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```




```
expr : Expr[Report] =
```

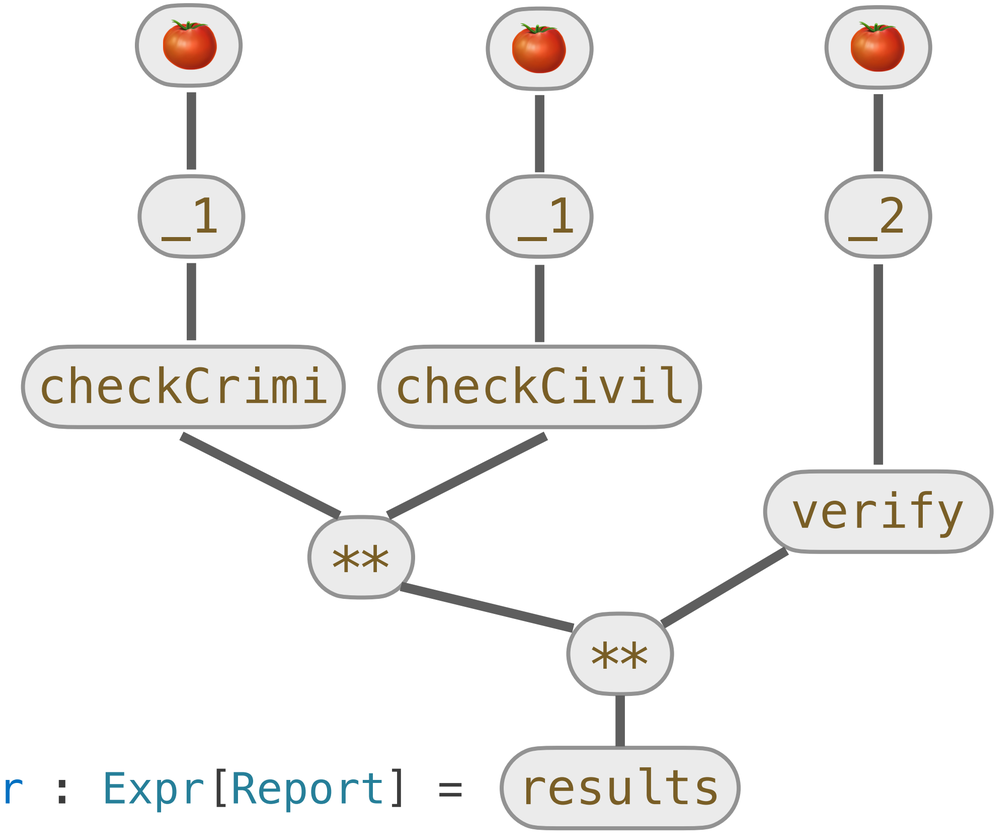
```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
val onAccept:  Expr[PersonalId ** Employmenti

  case       (id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

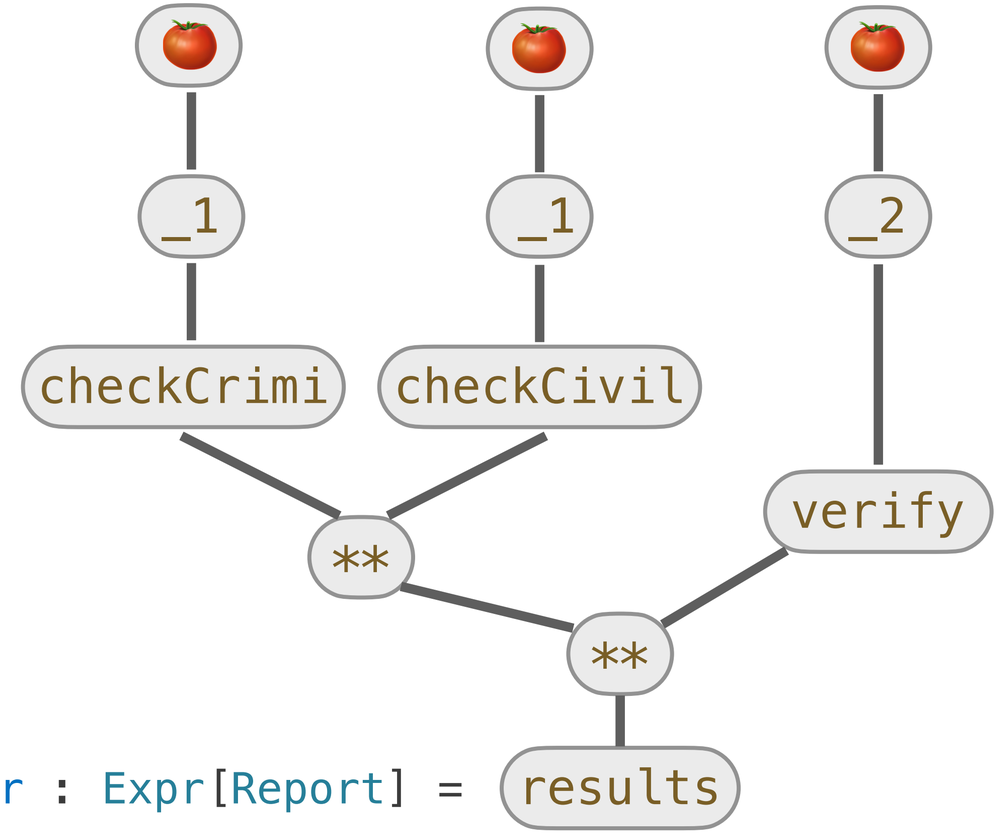
```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
val onA                                   Employmenti
  case
   val
   val
   val
   resu                                        f)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



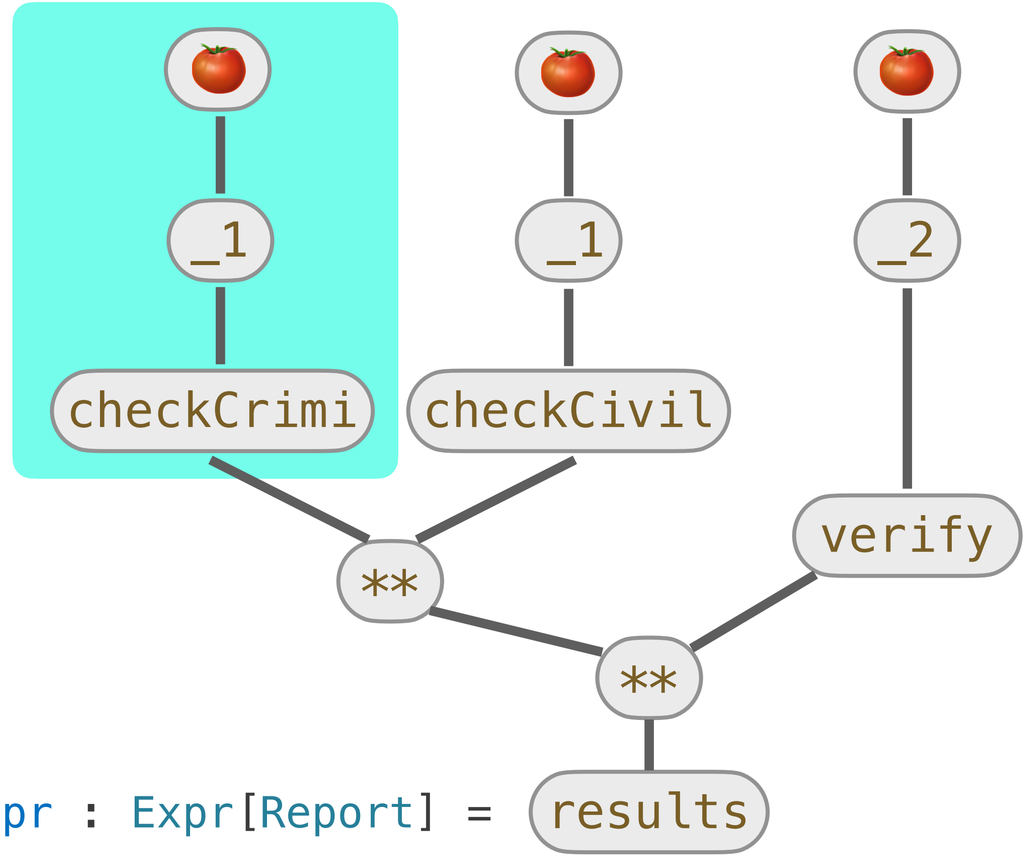

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
val onA                                   Employmenti
  case
   val
   val
   val
   resu                                        f)
```

```
_1 >>> checkCrimi
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



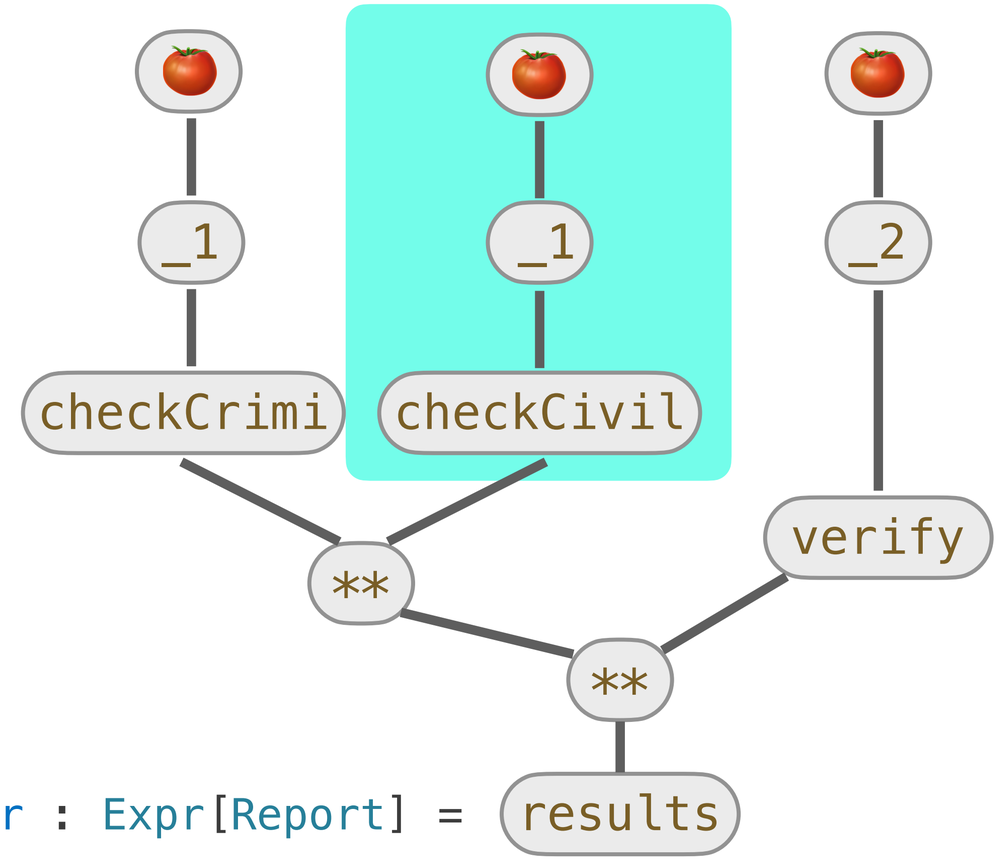

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
val onA                                   Employmenti
  case
   val
   val
   val
   resu                                        f)
```

```
_1 >>> checkCrimi
_1 >>> checkCivil
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



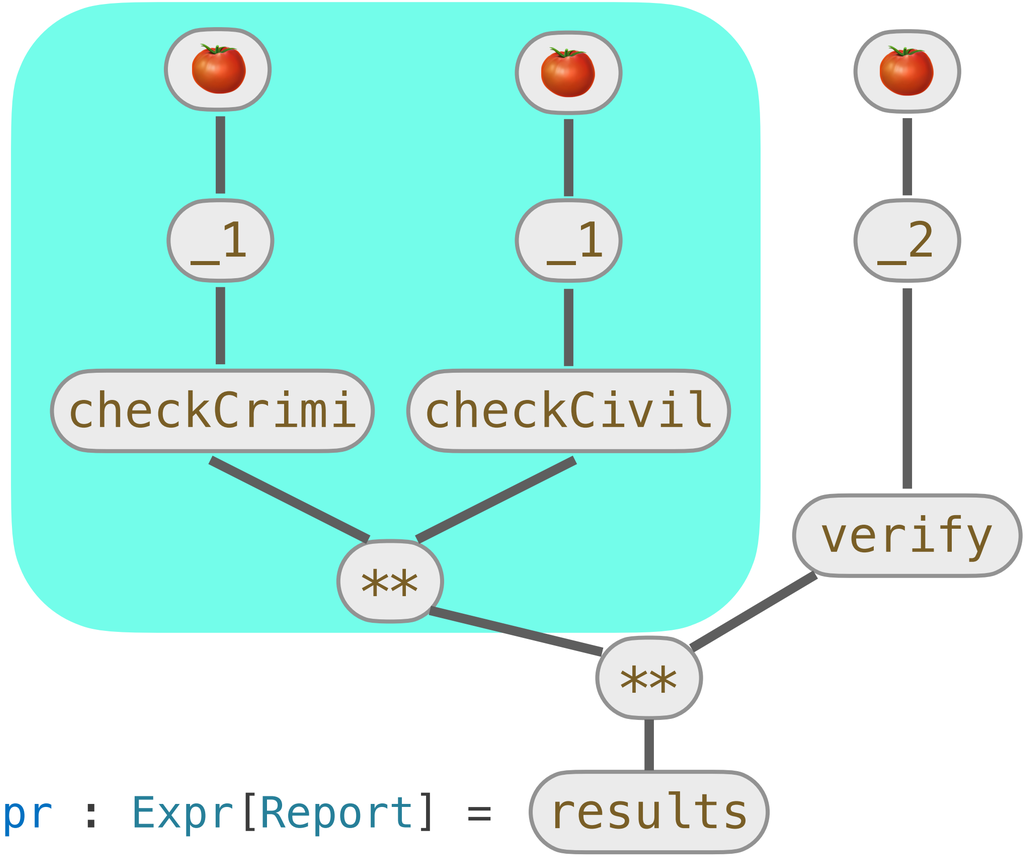

```
expr : Expr[Report] =
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
Dup() >>> Par(
  _1 >>> checkCrimi,
  _1 >>> checkCivil
)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
expr : Expr[Report] = results
```

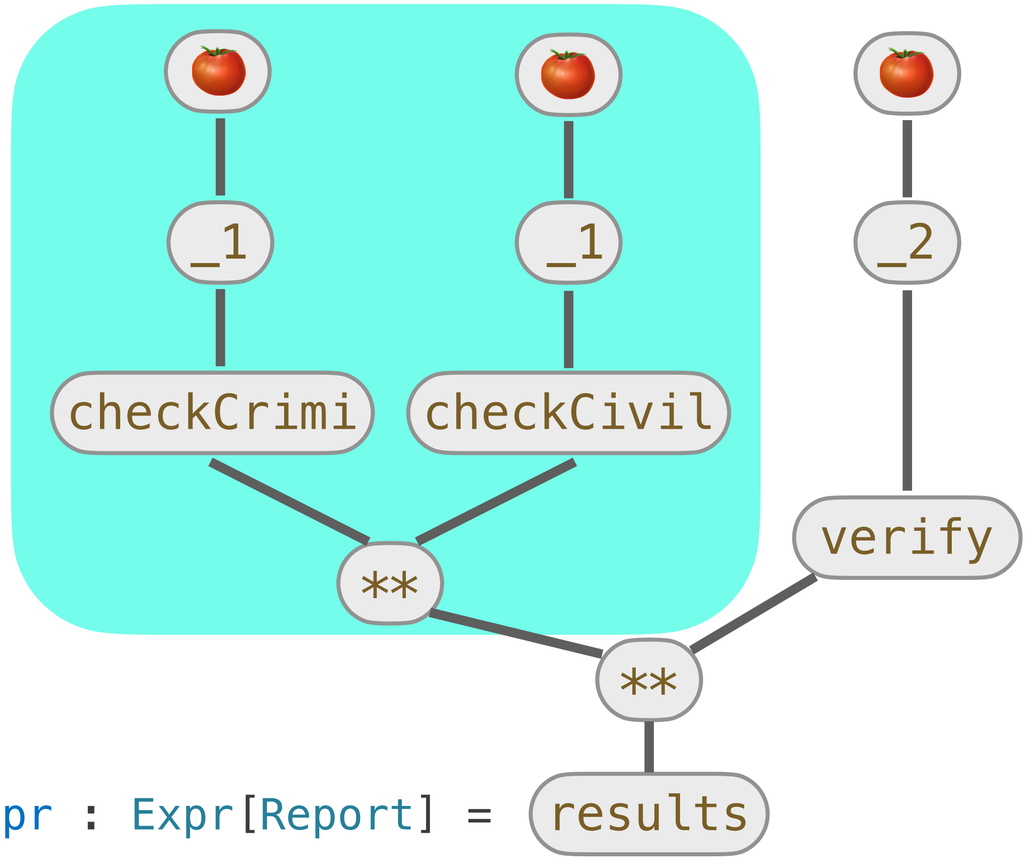



```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅  : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
Dup() >>> Par(
  _1 >>> checkCrimi,
  _1 >>> checkCivil
)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



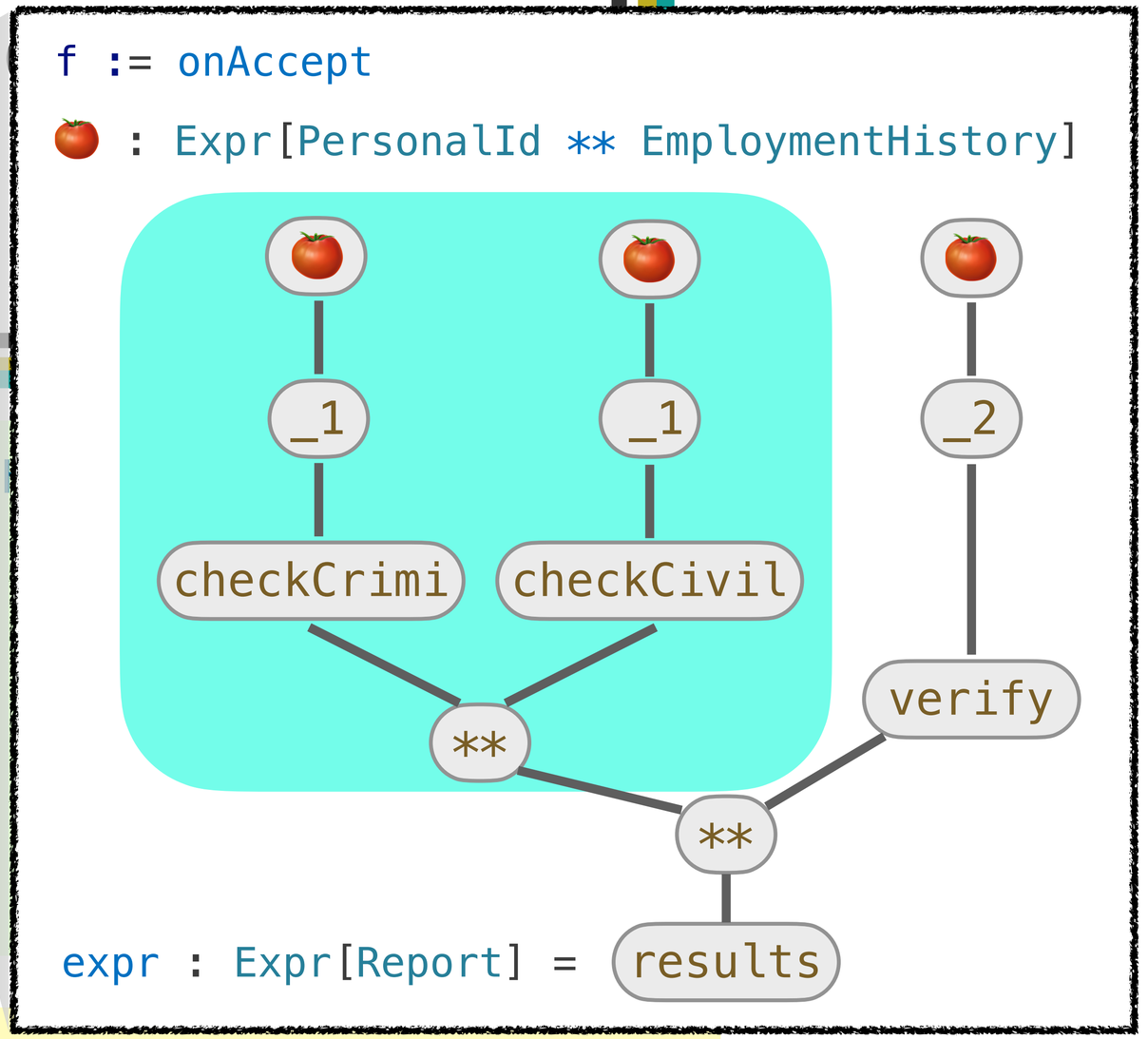

```
expr : Expr[Report] =
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
_1 >>> Dup() >>> Par(
  checkCrimi,
  checkCivil
)
```

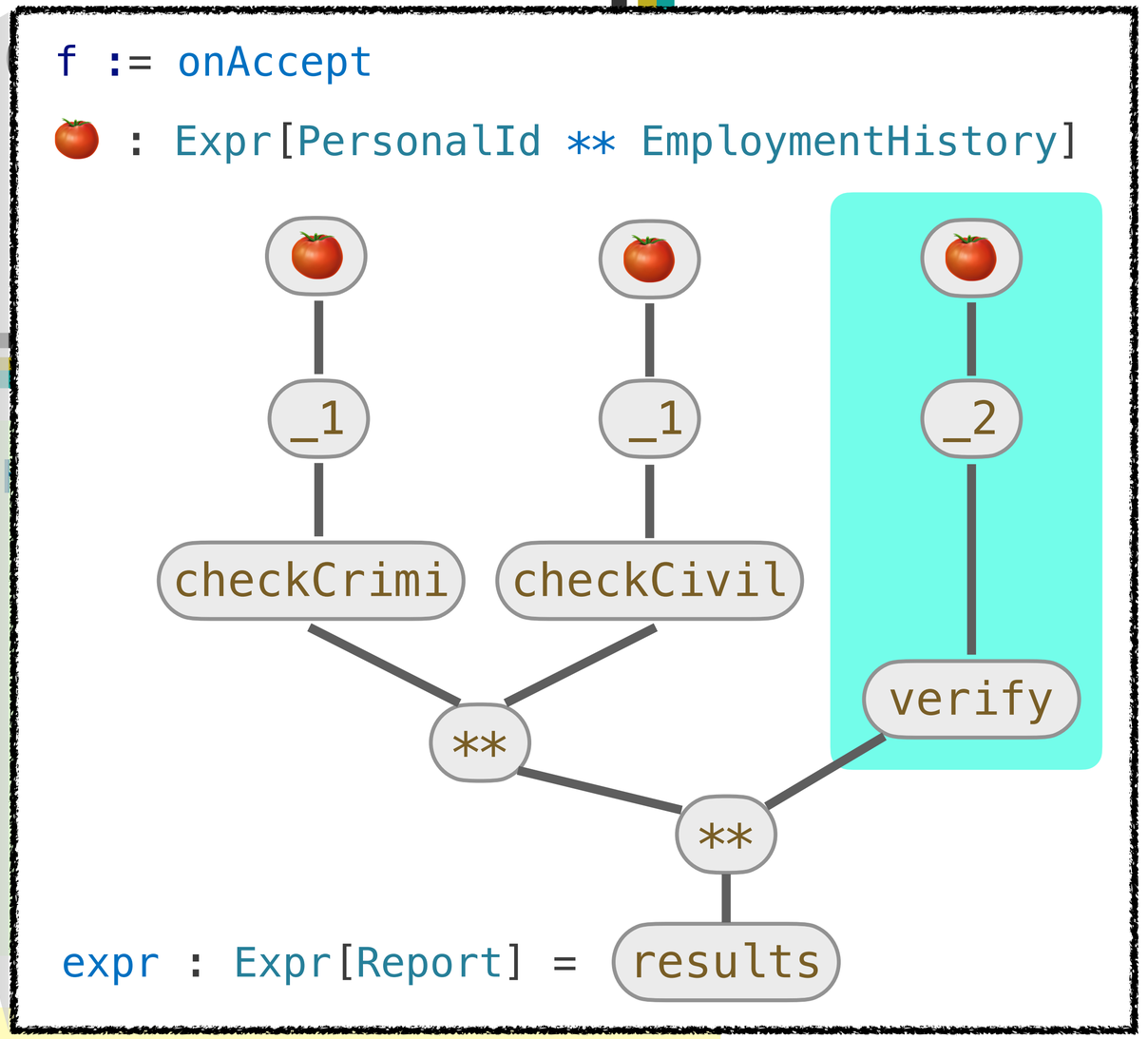
```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```



```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
val onA                                   Employmenti
  case
   val
   val
   val
   resu                                        .f)
```

```
_1 >>> Dup() >>> Par(
  checkCrimi,
  checkCivil
)
_2 >>> verify
```

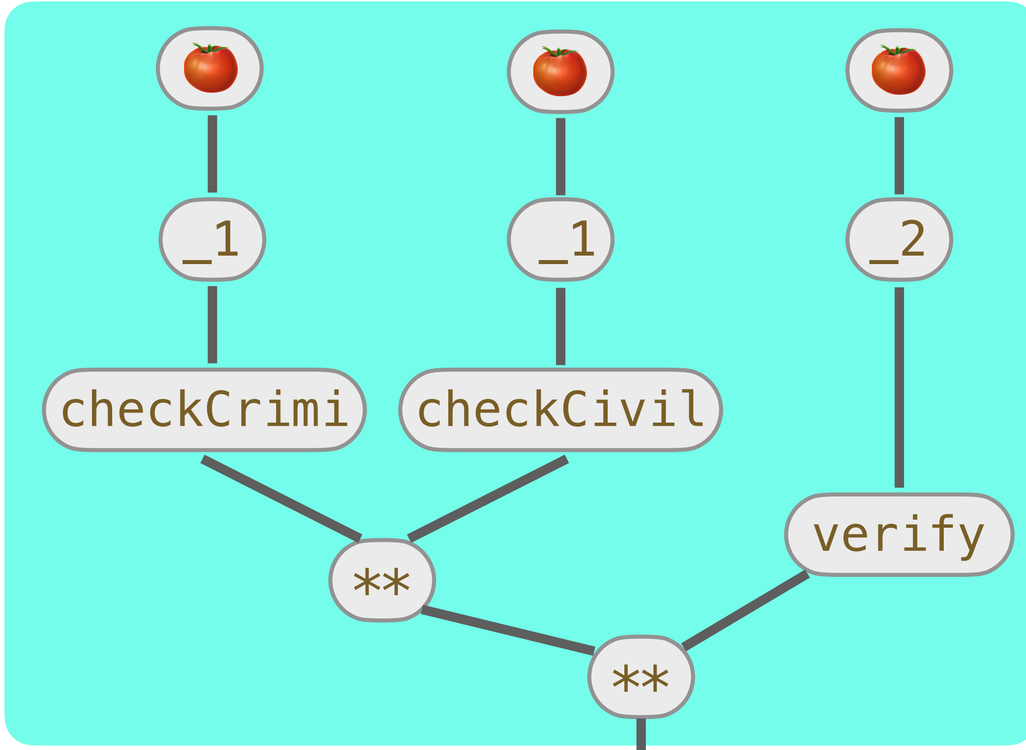



```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
Dup() >>> Par(
  _1 >>> Dup() >>> Par(
    checkCrimi,
    checkCivil
  ),
  _2 >>> verify
)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
expr : Expr[Report] = results
```

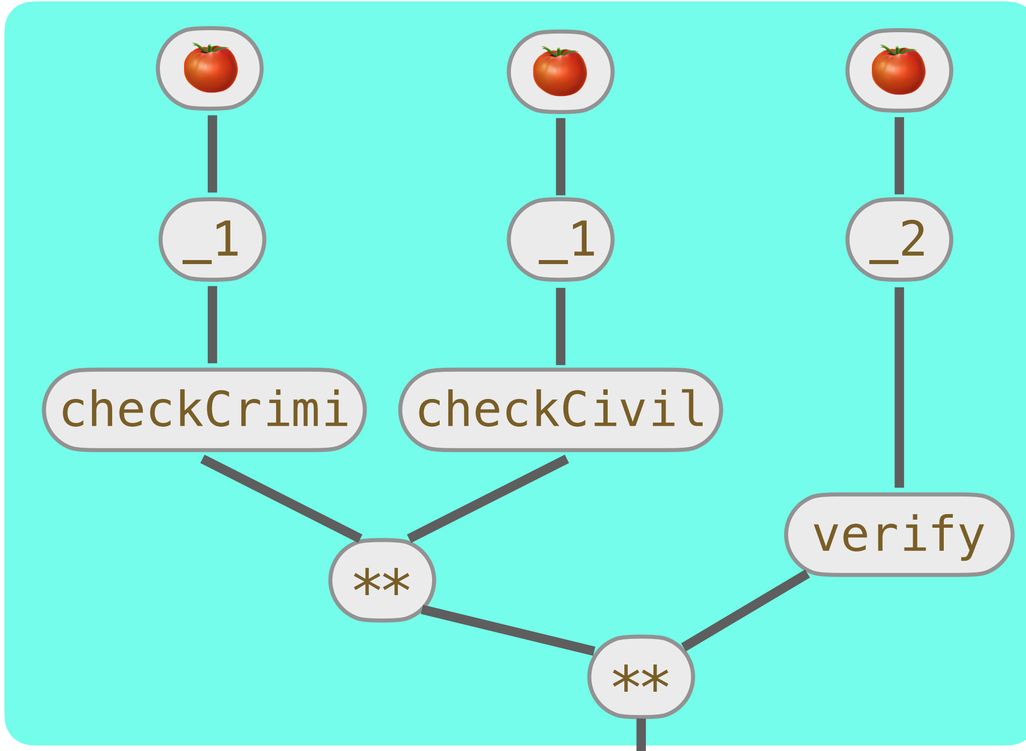



```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
Dup() >>> Par(
  _1 >>> Dup() >>> Par(
    checkCrimi,
    checkCivil
  ),
  _2 >>> verify
)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
expr : Expr[Report] = results
```

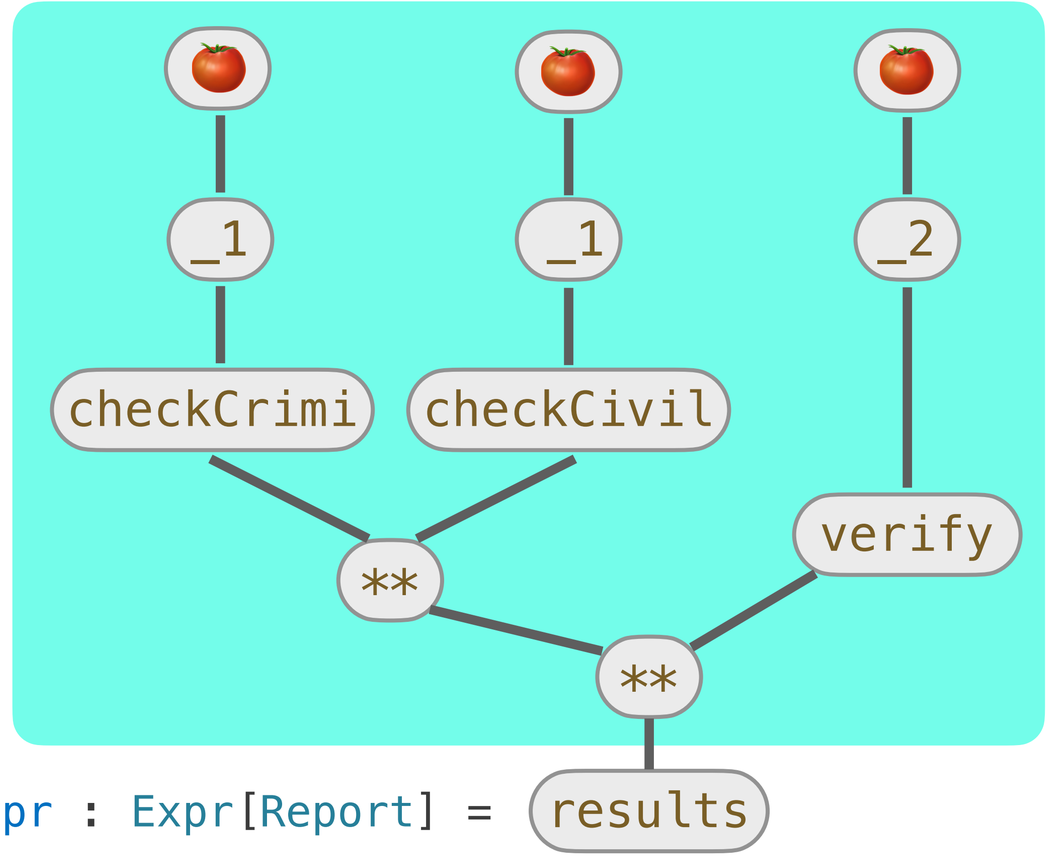



```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
Par(
  Dup() >>> Par(
    checkCrimi,
    checkCivil
  ),
  verify
)
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



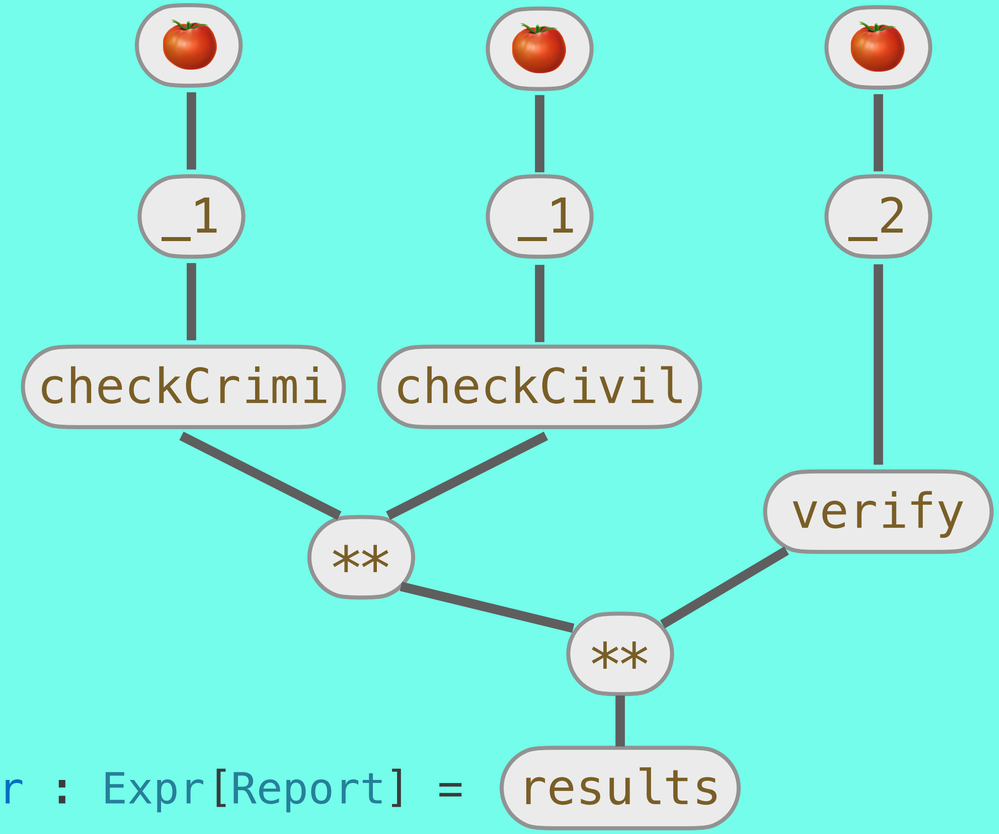

```
expr : Expr[Report] =
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

```
def delambdify[A, B](f: Expr[A] => Expr[B]): Flow[A, B] =
  val 🍅 : Expr[A] = freshVariable
  val expr : Expr[B] = f(🍅)
  eliminate(🍅, from = expr)
```

```
Par(
  Dup() >>> Par(
    checkCrimi,
    checkCivil
  ),
  verify
) >>> results
```

```
f := onAccept
🍅 : Expr[PersonalId ** EmploymentHistory]
```



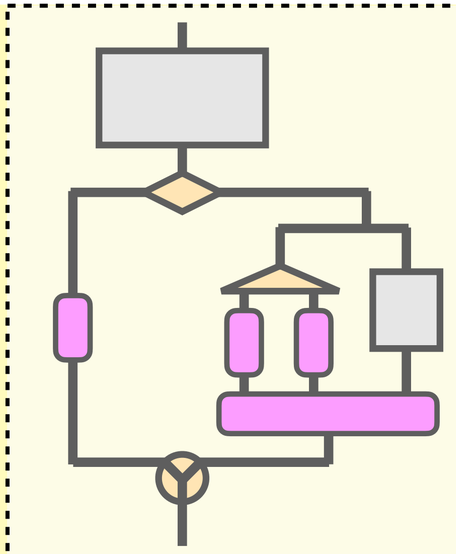

```
expr : Expr[Report] =
```

```
delambdify(onAccept): Flow[PersonalId ** EmploymentHistory, Report]
```

… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

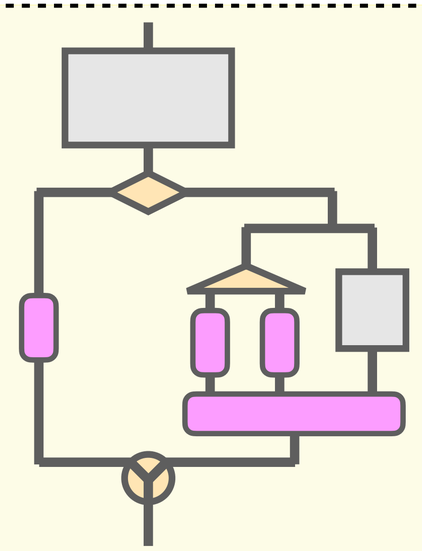

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

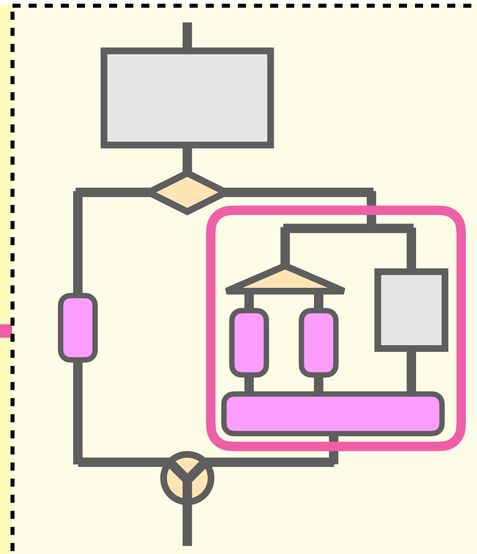

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

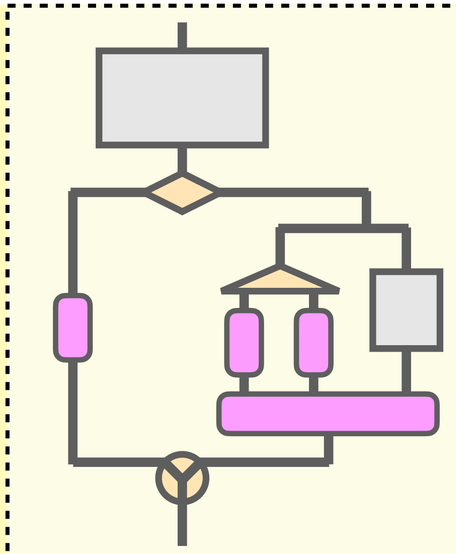

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

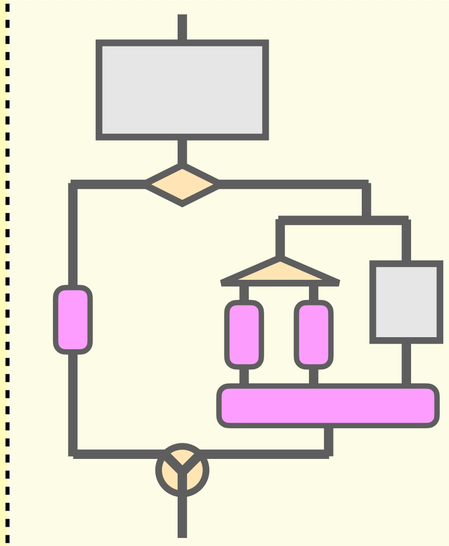

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

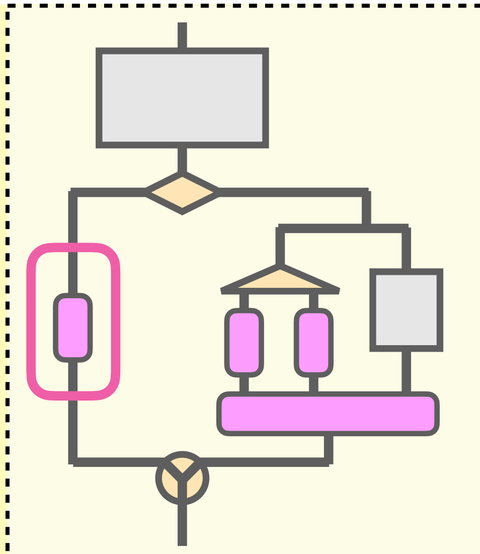

```
AndThen(
 askForAccept,
 Switch(
  declined,
   AndThen(
    Par(
     AndThen(
      Dup(),
      Par(checkCrimi, checkCivil)
     ),
     verify
    ),
    results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

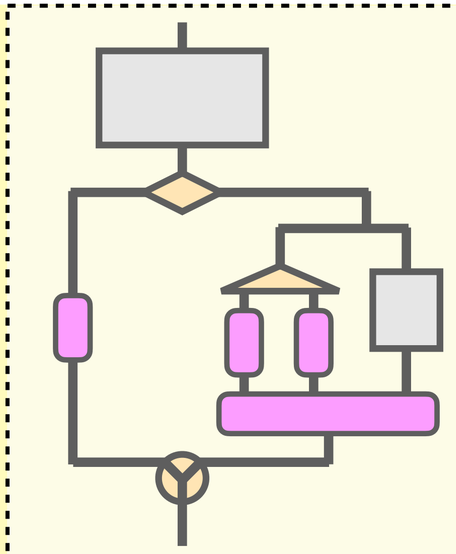

```
AndThen(
 askForAccept,
 Switch(
  declined,
   AndThen(
    Par(
     AndThen(
      Dup(),
      Par(checkCrimi, checkCivil)
     ),
     verify
    ),
    results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

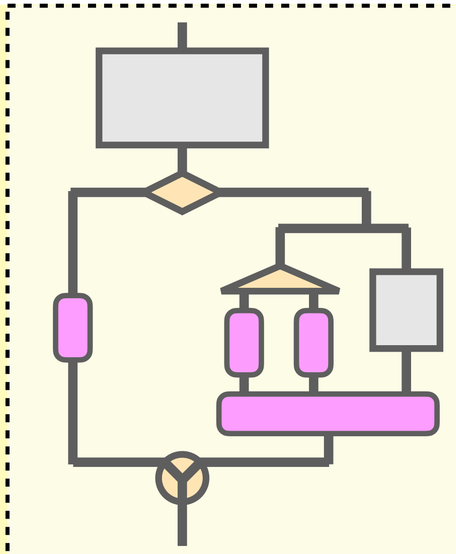

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```


```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



`switch` : the easy case

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

`switch` : the easy case

non-capturing

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

`switch` : the easy case

non-capturing

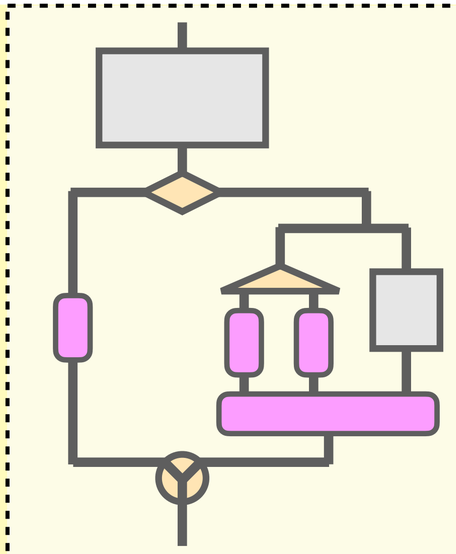
```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

```
extension [A, B](expr: Expr[A ++ B])
  def switch[R](
    f: Either[Expr[A], Expr[B]] => Expr[R],
  ): Expr[R] =
    val f1: Flow[A, R] = delambdify(a => f(Left(a)))
    val f2: Flow[B, R] = delambdify(b => f(Right(b)))
    val ff: Flow[A ++ B, R] = Switch(f1, f2)
    ff(expr)
```

… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

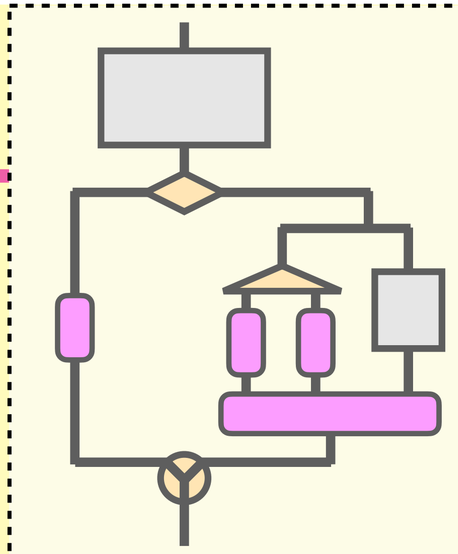

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```

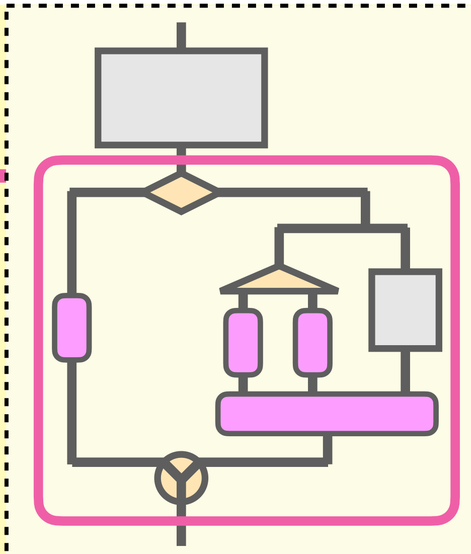

```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



… delambdified so far …

```
Flow { candidate =>
 askForAccept(candidate) switch {
  case Left(x) =>
   declined(x)
  case Right(id ** history) =>
   val crimi = checkCrimi(id)
   val civil = checkCivil(id)
   val verif = verify(history)
   results(crimi ** civil ** verif)
 }
}
```


```
AndThen(
 askForAccept,
 Switch(
  declined,
  AndThen(
   Par(
    AndThen(
     Dup(),
     Par(checkCrimi, checkCivil)
    ),
    verify
   ),
   results
)))
```



`switch`: the capturing case

```
def switch[A, B, …, R](
```

`switch`: the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:      [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:      [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
```

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:      [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
```

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:      [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
```

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

```
case Distribute[X, Y, Z]()
extends Flow[
  X ** (Y ++ Z),
  (X ** Y) ++ (X ** Z)
]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:     [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
): (
```

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

```
case Distribute[X, Y, Z]()
extends Flow[
  X ** (Y ++ Z),
  (X ** Y) ++ (X ** Z)
]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:     [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
): (
  Expr[Q],
```

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

```
case Distribute[X, Y, Z]()
extends Flow[
  X ** (Y ++ Z),
  (X ** Y) ++ (X ** Z)
]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:      [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
): (
  Expr[Q],
```

Captured

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

```
case Distribute[X, Y, Z]()
extends Flow[
  X ** (Y ++ Z),
  (X ** Y) ++ (X ** Z)
]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:      [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
): (
  Expr[Q],
  Flow[Q ** (A ++ B ++ …), R]
```

Captured

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

```
case Distribute[X, Y, Z]()
extends Flow[
  X ** (Y ++ Z),
  (X ** Y) ++ (X ** Z)
]
```

`switch` : the capturing case

```
def switch[A, B, …, R](
  fa: Expr[A] => Expr[R],
  fb: Expr[B] => Expr[R],
  …
  sum:     [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R],
  distrib: [X,Y,Z] => Flow[X ** (Y ++ Z), (X ** Y) ++ (X ** Z)]
): (
  Expr[Q],
  Flow[Q ** (A ++ B ++ …), R]
) // for some type Q
```

```
case Switch[X, Y, R](
  l: Flow[X, R],
  r: Flow[Y, R],
) extends Flow[X ++ Y, R]
```

```
case Distribute[X, Y, Z]()
extends Flow[
  X ** (Y ++ Z),
  (X ** Y) ++ (X ** Z)
]
```

Captured

Also Applicable To

- Loops

```
case DoWhile[A, B]( iteration: Flow[A, A ++ B] ) extends Flow[A, B]
```

- Recursion
  - incl. recursive types
- Higher-order functions
  - Including closures
- Linearly typed DSLs

… status so far …

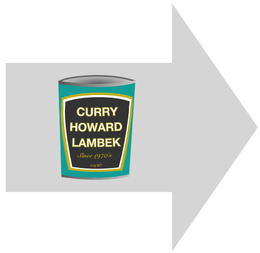
… status so far …

```
Flow { x =>
 ... switch {
  case ... =>
   ...
  case ... =>
   val y = ...
   ...
 }
}
```

… status so far …

```
Flow { x =>
 ... switch {
  case ... =>
   ...
  case ... =>
   val y = ...
   ...
 }
}
```

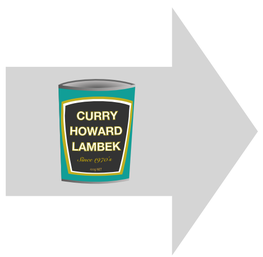
code



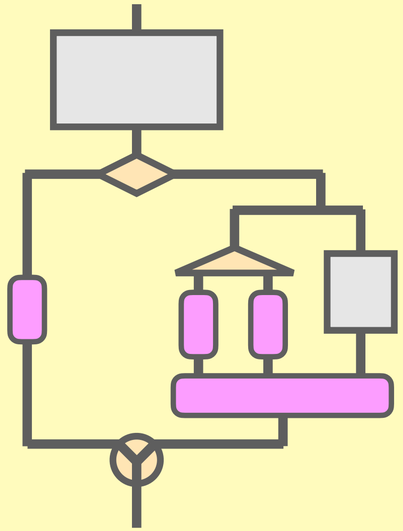
… status so far …

```
Flow { x =>
 ... switch {
  case ... =>
   ...
  case ... =>
   val y = ...
   ...
 }
}
```

code




```
AndThen(
 ...
 Switch(
  ...
  Par(
   Dup()...
   ...
  ),
))
```

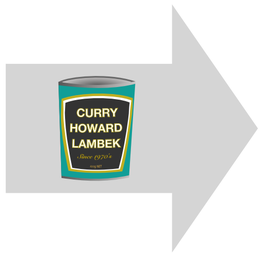


… as data

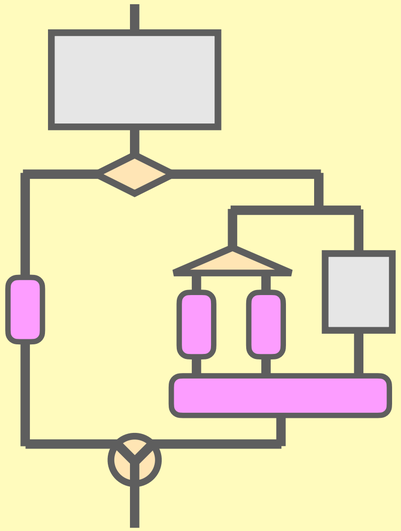
… status so far …

```
Flow { x =>
 ... switch {
  case ... =>
   ...
  case ... =>
   val y = ...
   ...
 }
}
```

code




```
AndThen(
 ...
 Switch(
  ...
  Par(
   Dup()...
   ...
  ),
))
```

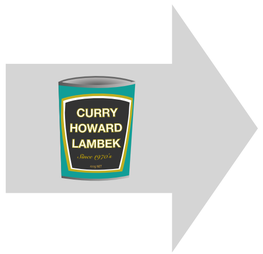


… as data

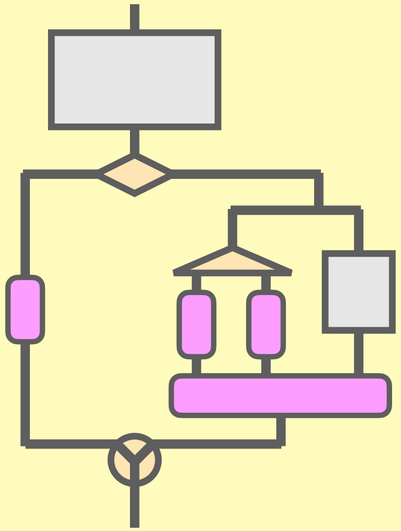
… status so far …

```
Flow { x =>
  ... switch {
   case ... =>
    ...
   case ... =>
    val y = ...
    ...
  }
}
```

code




```
AndThen(
  ...
  Switch(
    ...
    Par(
     Dup()...
     ...
    ),
))
```



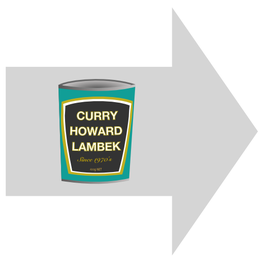
… as data

?

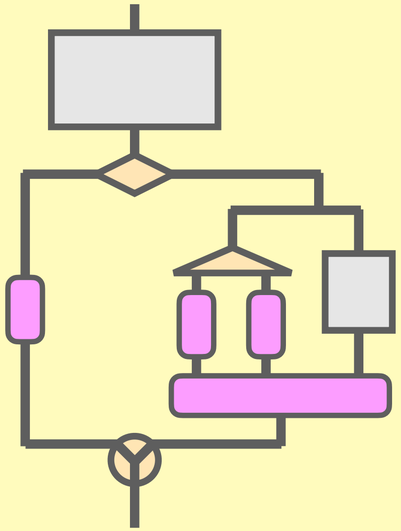
… status so far …

```
Flow { x =>
  ... switch {
   case ... =>
    ...
   case ... =>
    val y = ...
    ...
  }
}
```

code




```
AndThen(
  ...
  Switch(
    ...
    Par(
     Dup()...
     ...
    ),
))
```



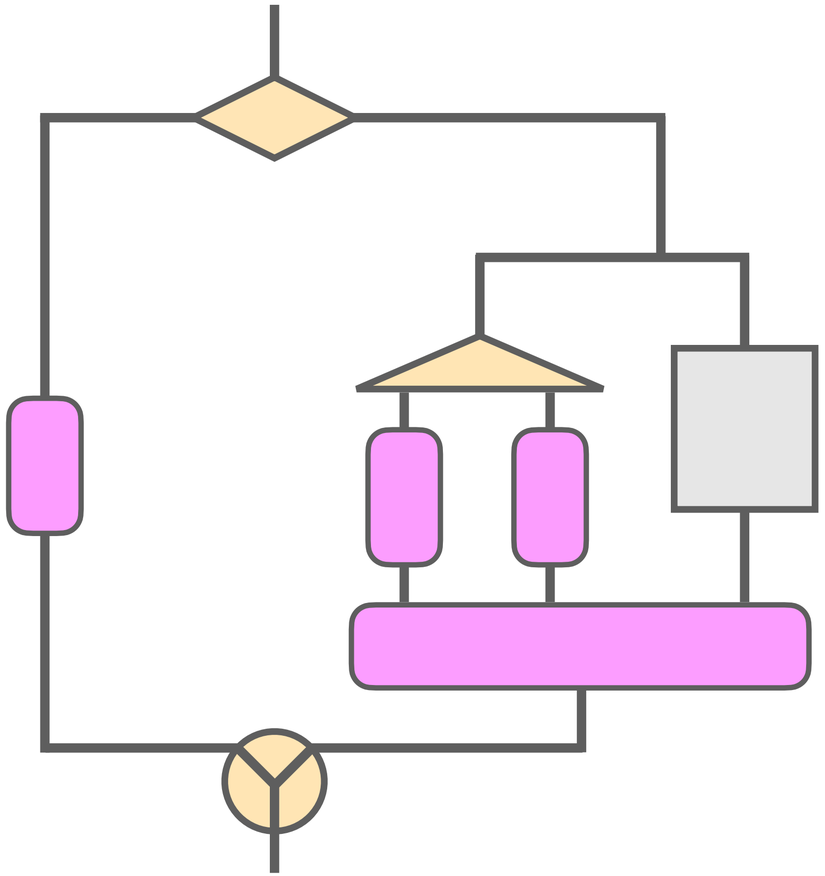
… as data

?

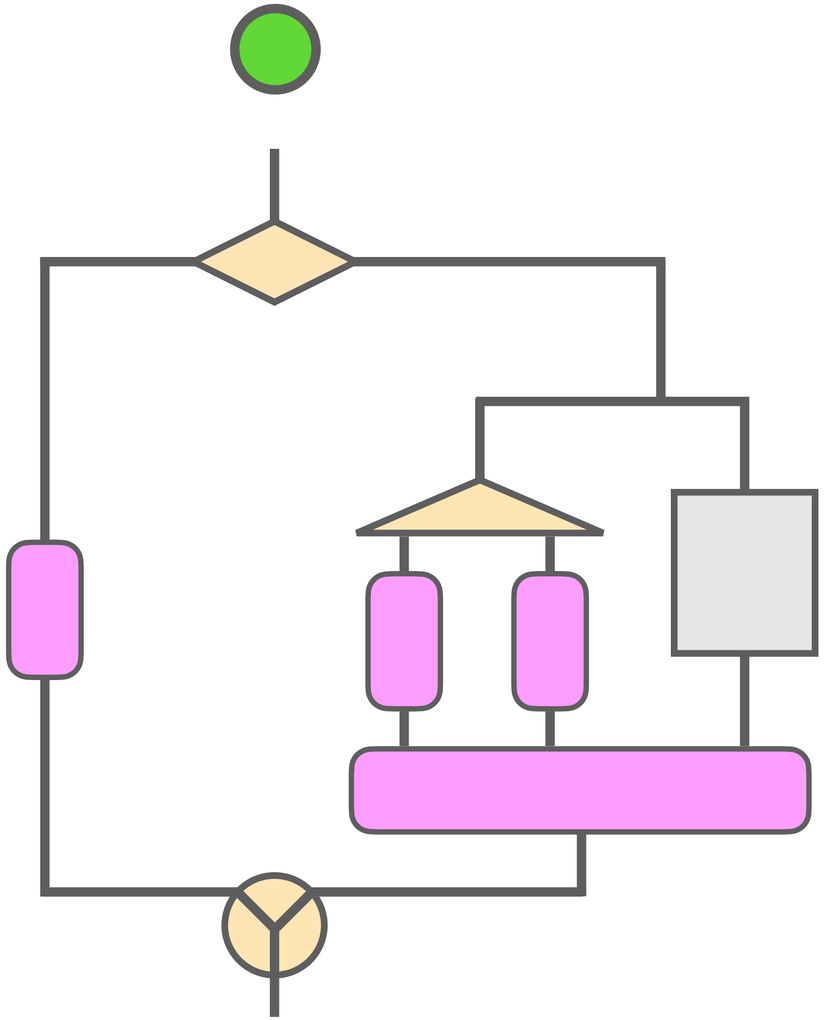
Where’s the *(durable) execution?*

Durable Execution

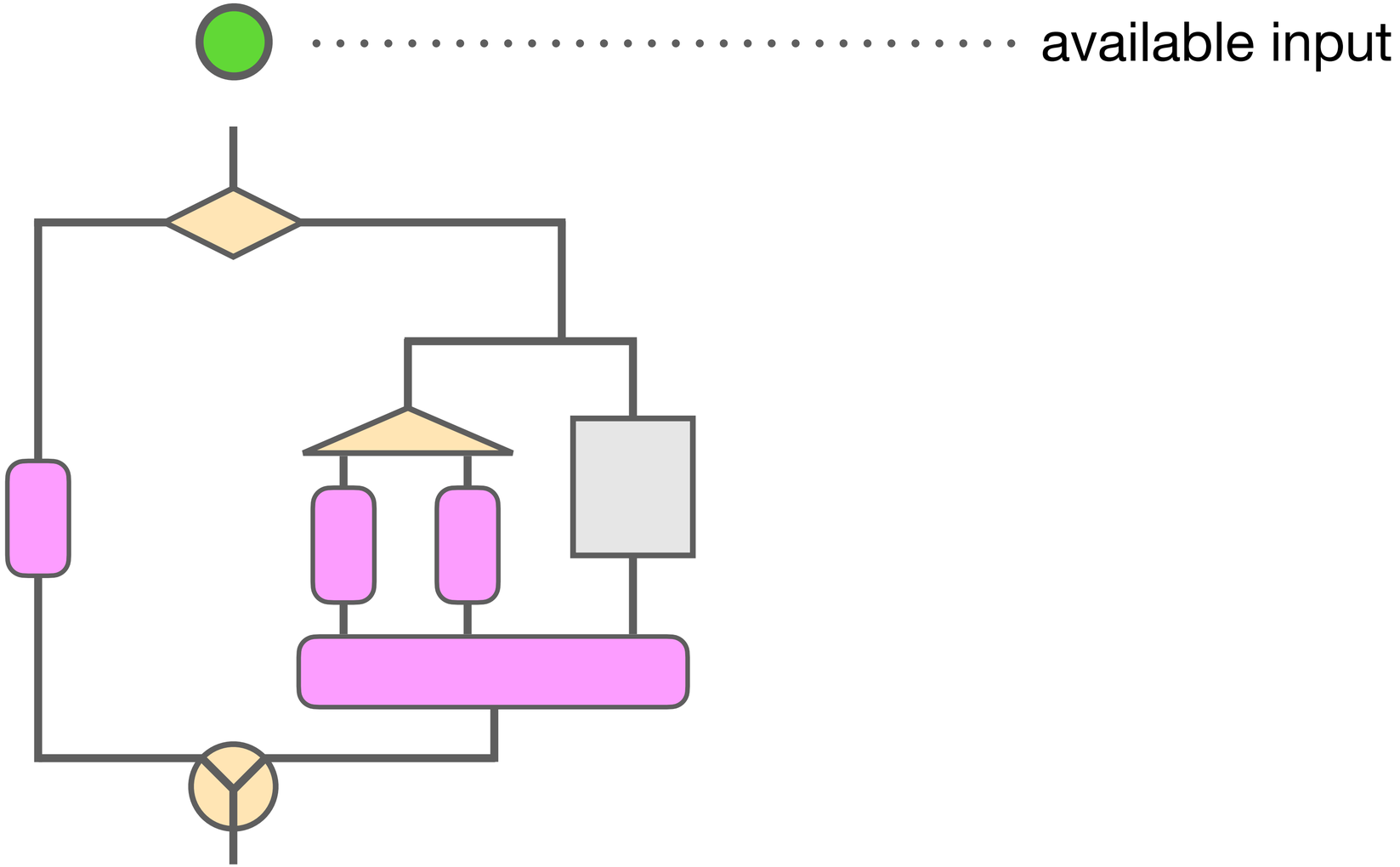
Durable Execution



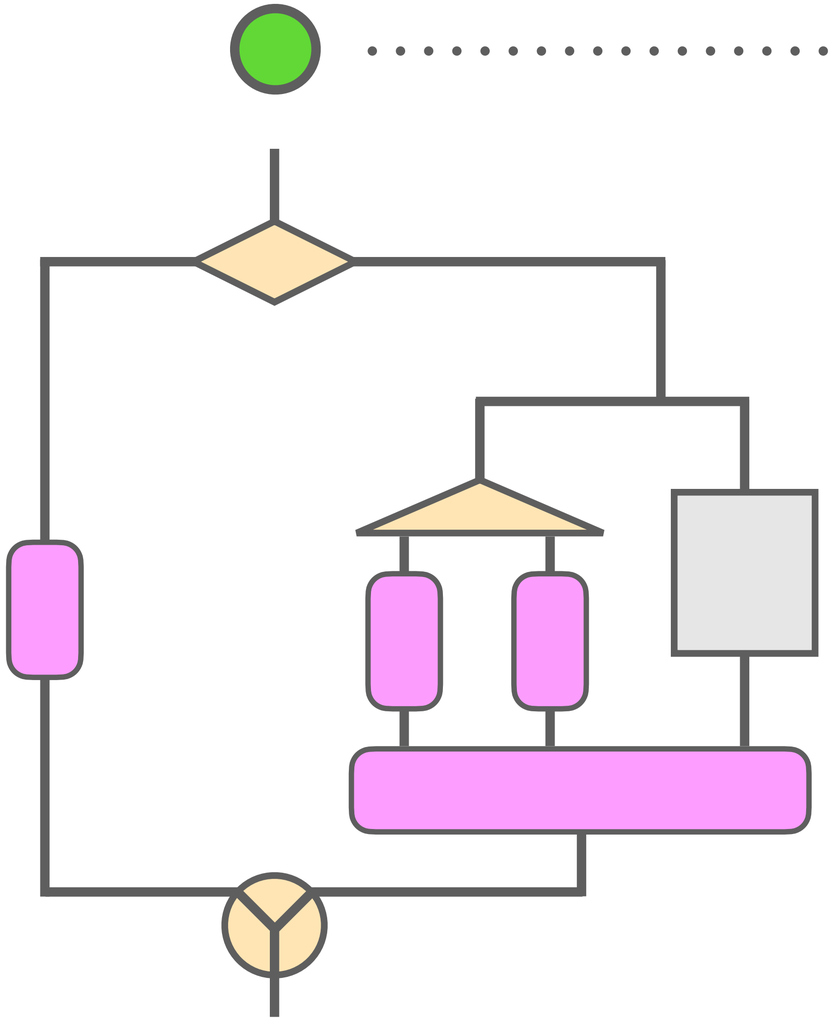
Durable Execution



Durable Execution



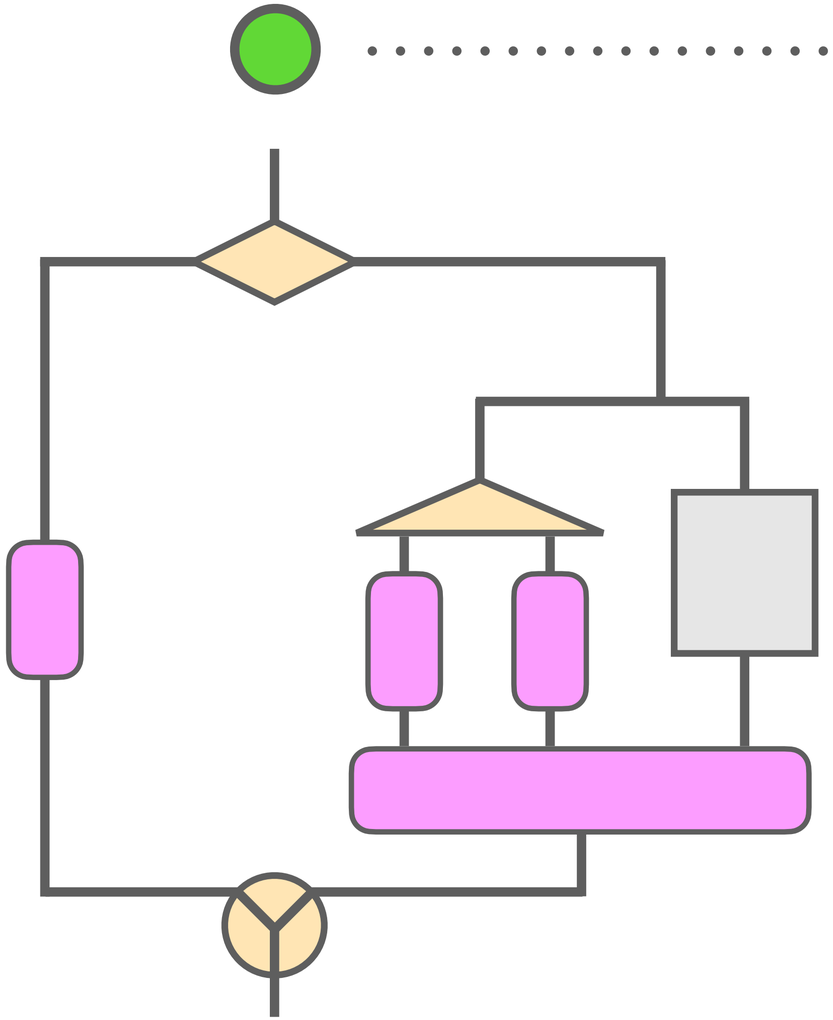
Durable Execution



available input

- data

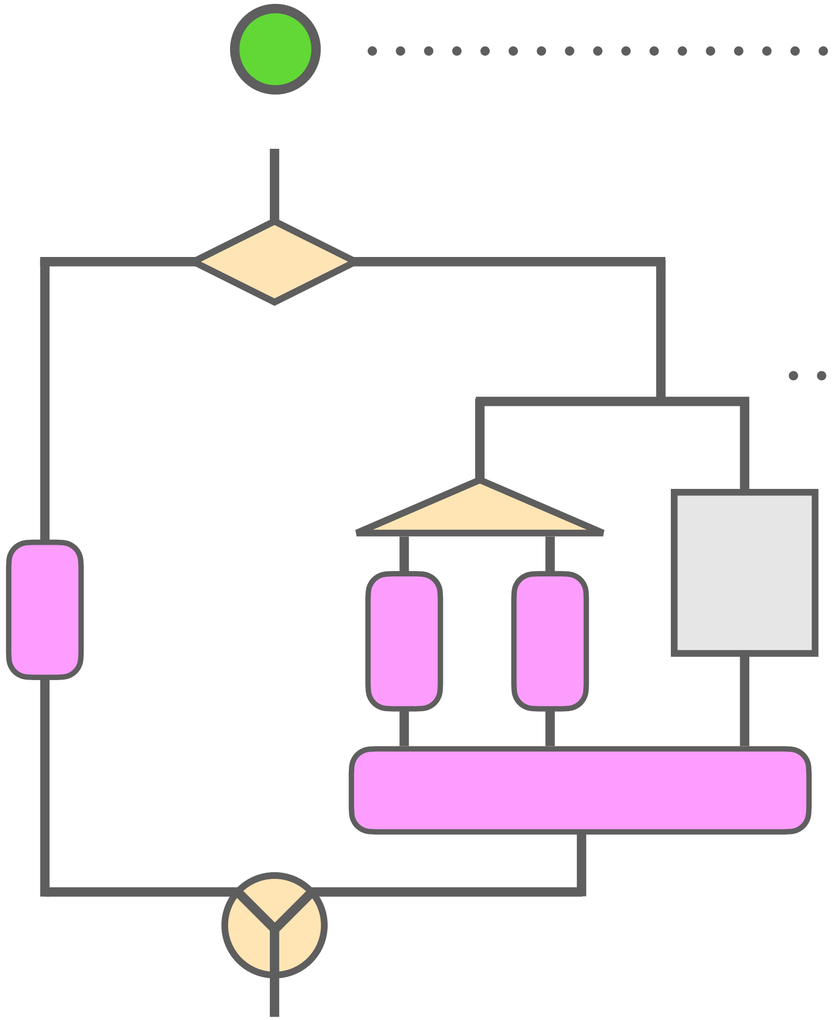
Durable Execution



available input

- data
- persistable

Durable Execution

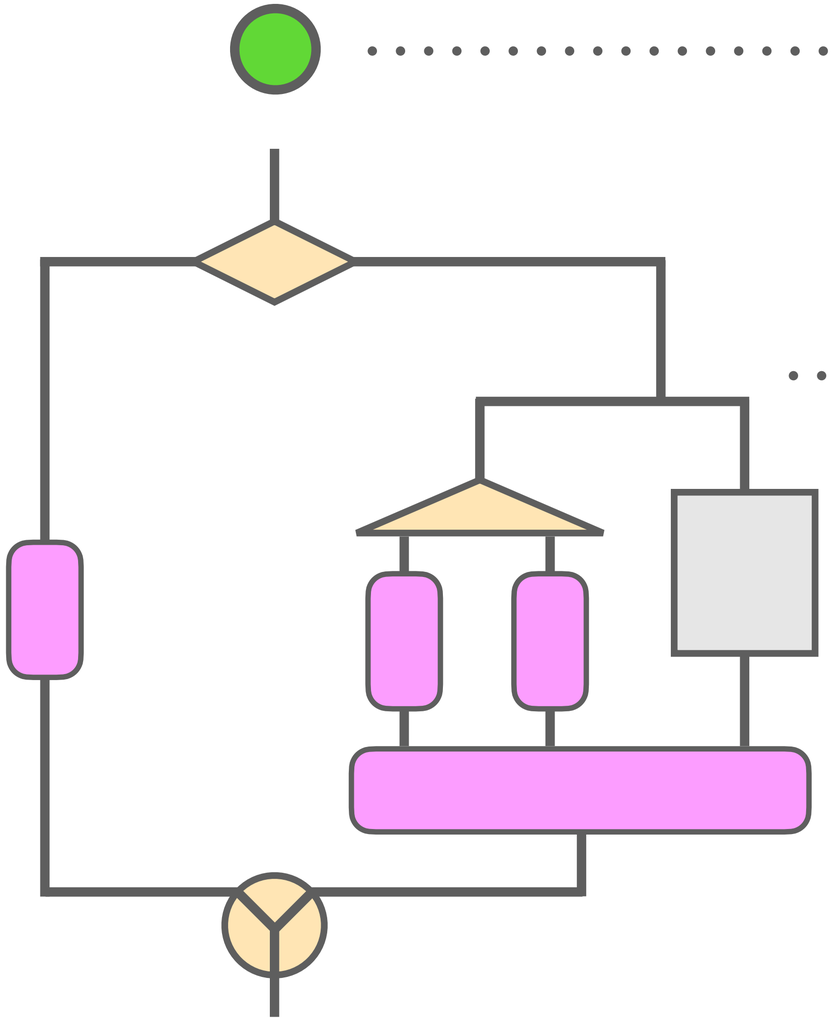


available input

- data
- persistable

continuation

Durable Execution



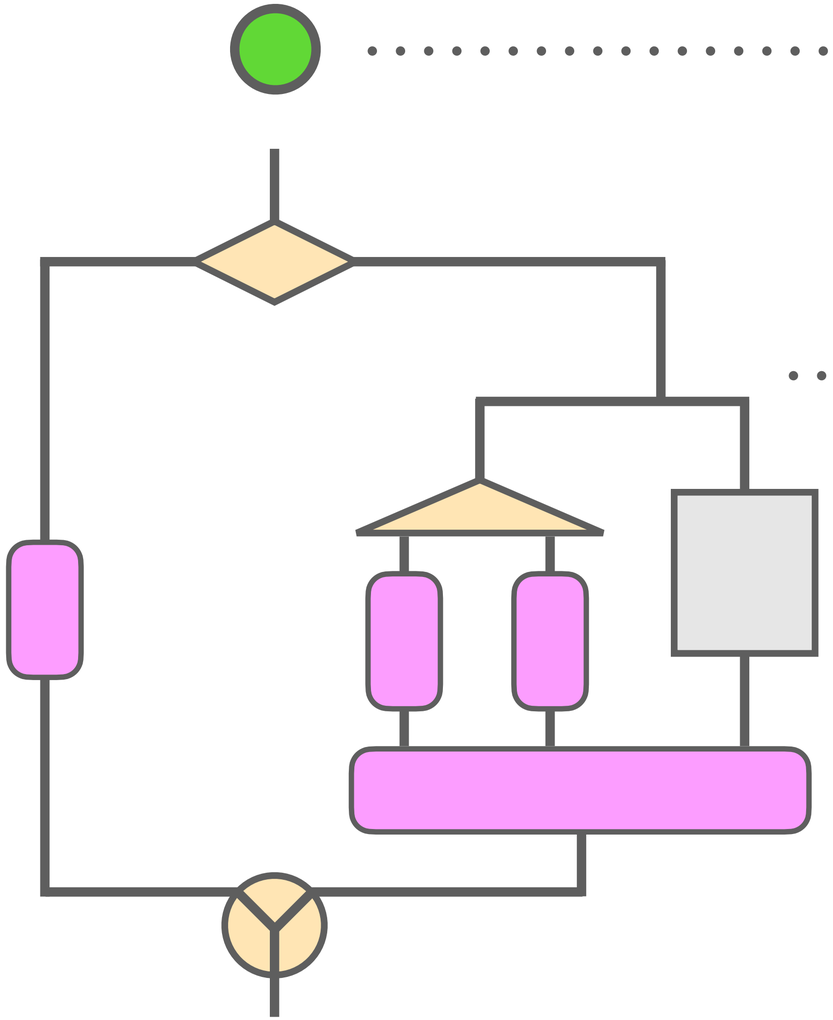
available input

- data
- persistable

continuation

- data

Durable Execution



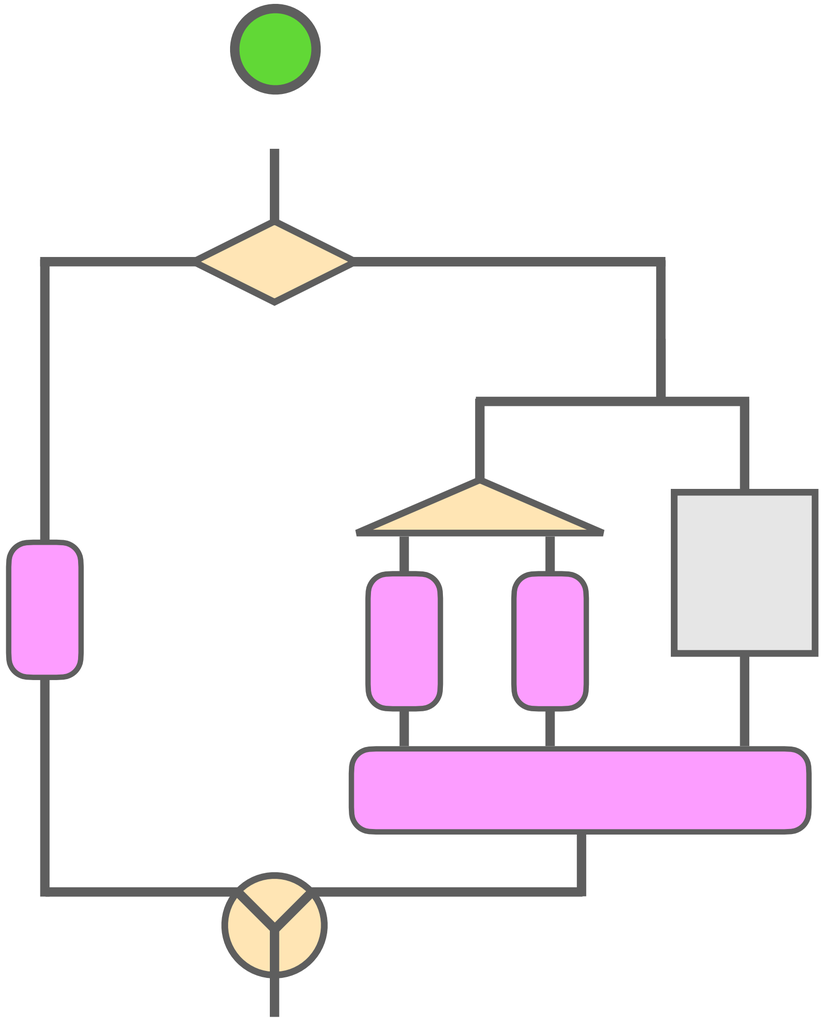
available input

- data
- persistable

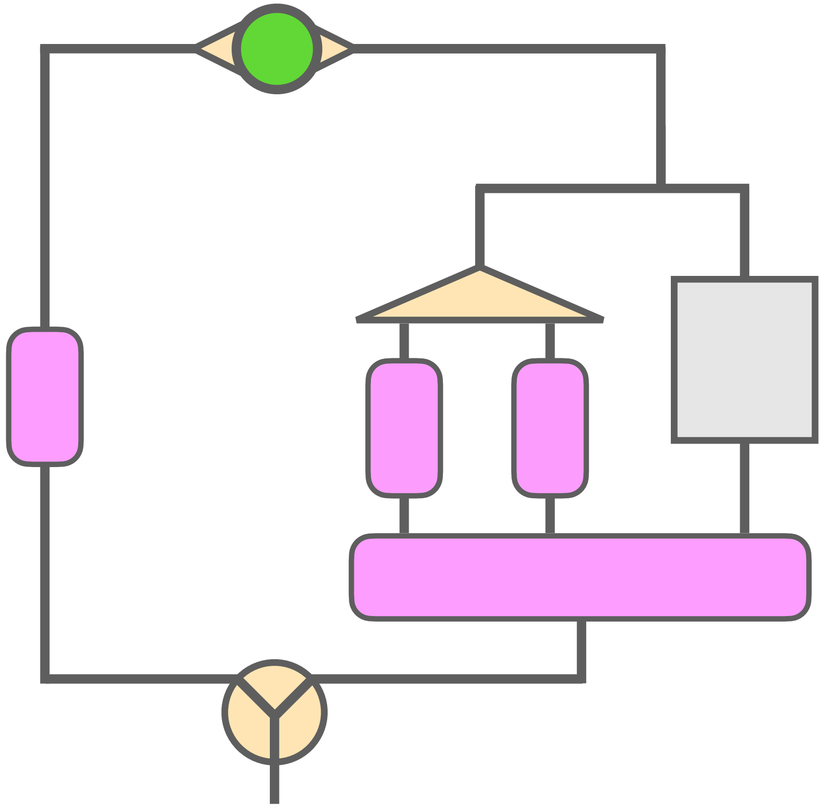
continuation

- data
- persistable

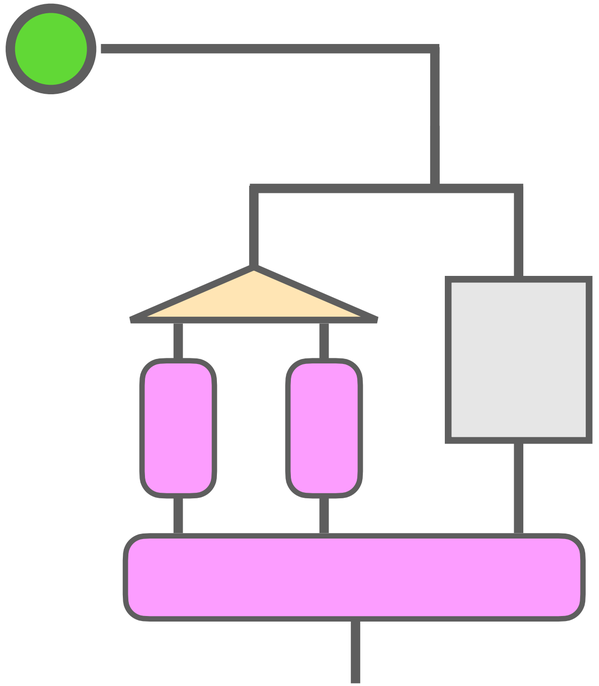
Durable Execution



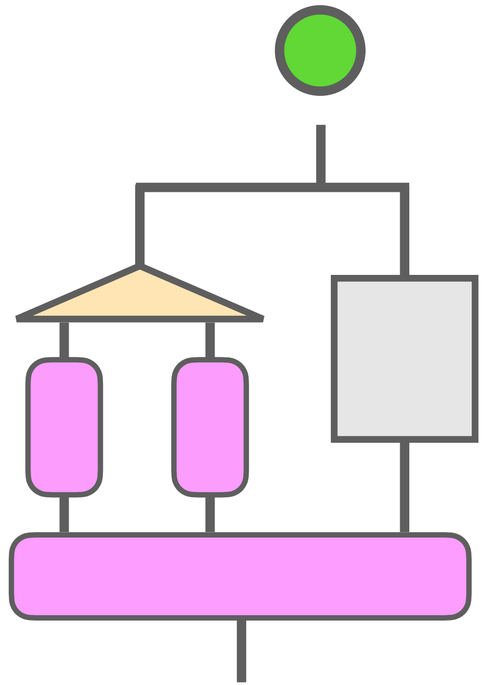
Durable Execution



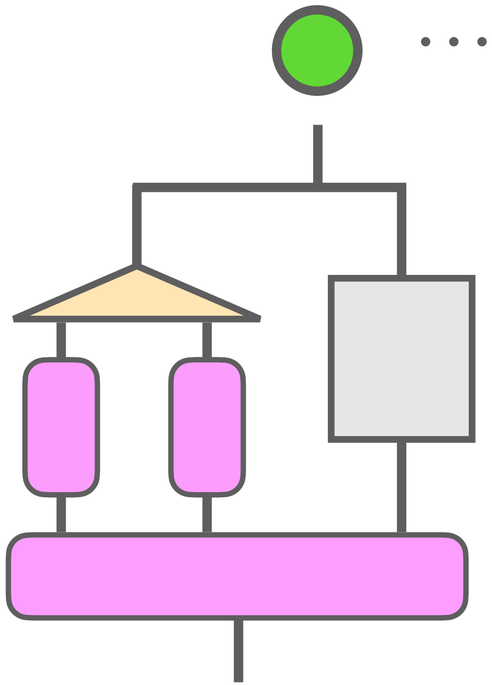
Durable Execution



Durable Execution



Durable Execution



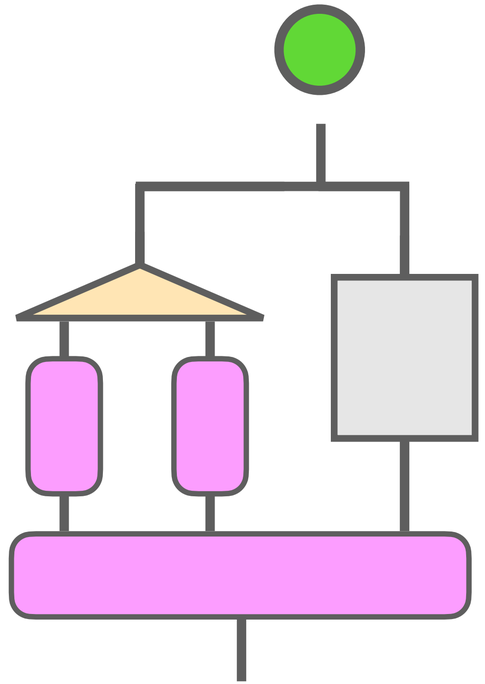
available input

- data
- persistable

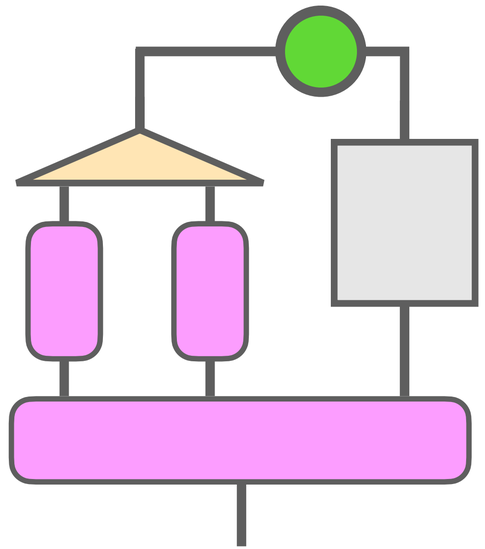
continuation

- data
- persistable

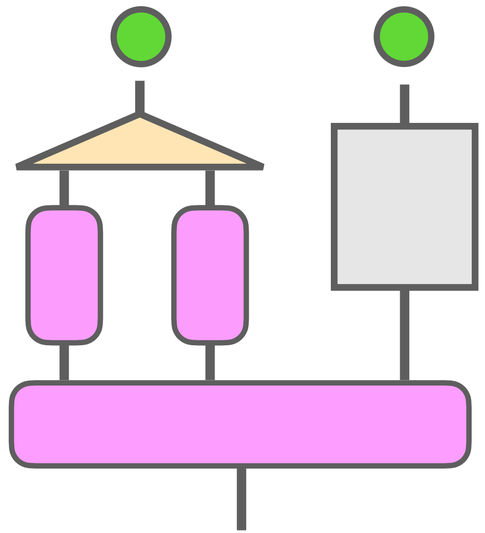
Durable Execution



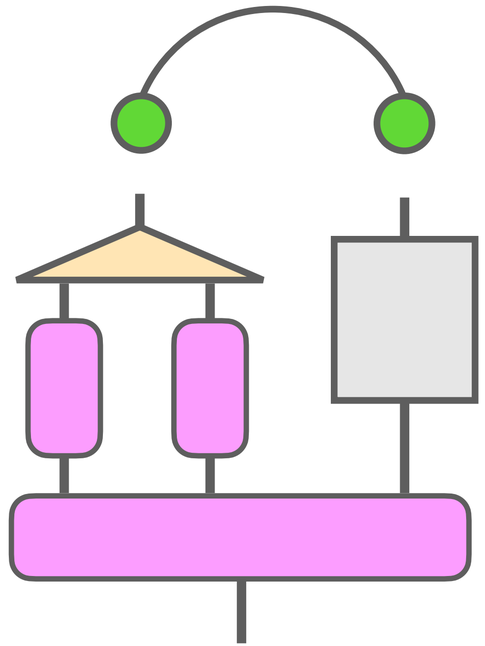
Durable Execution



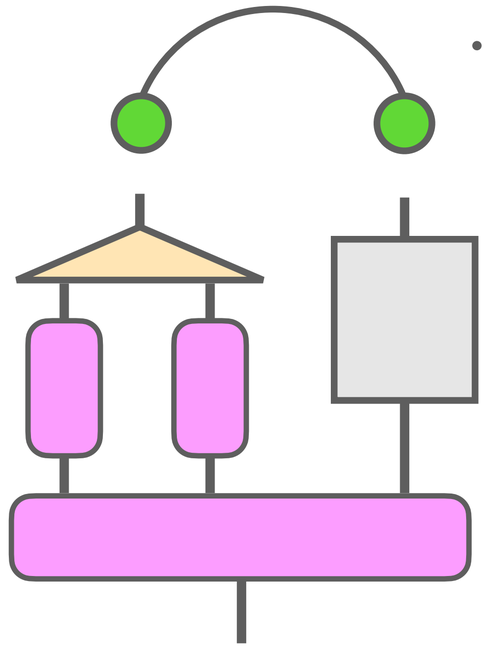
Durable Execution



Durable Execution



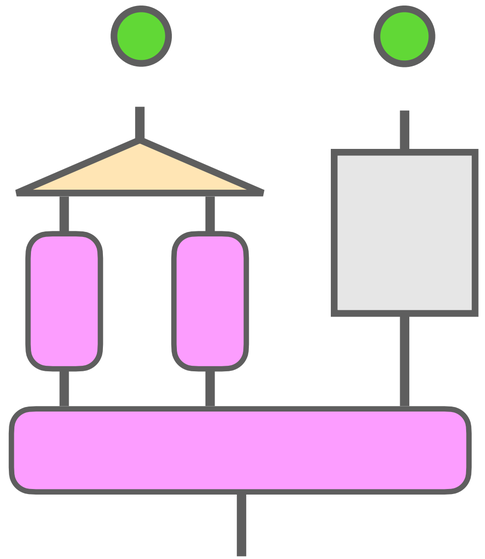
Durable Execution



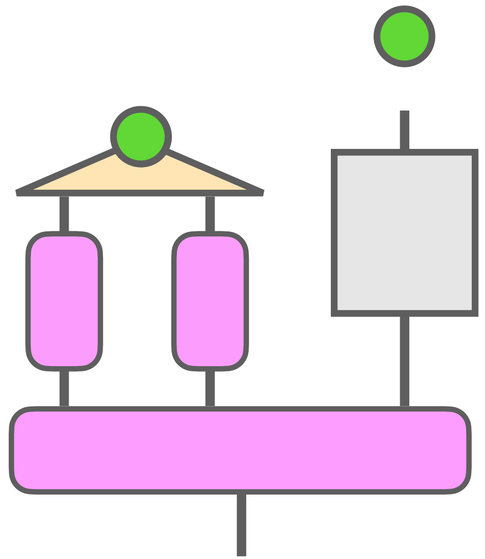
available inputs
- data
- persistable

continuation
- data
- persistable

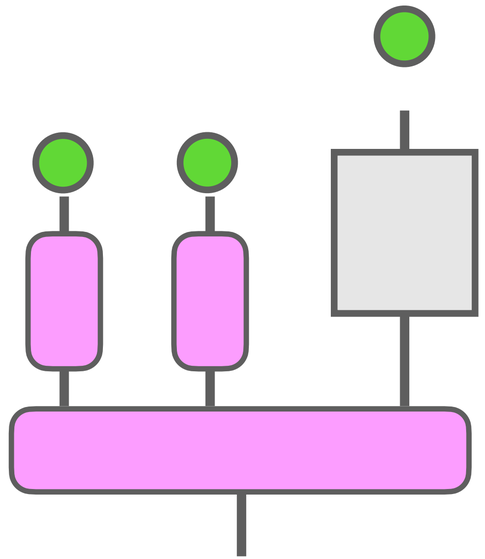
Durable Execution



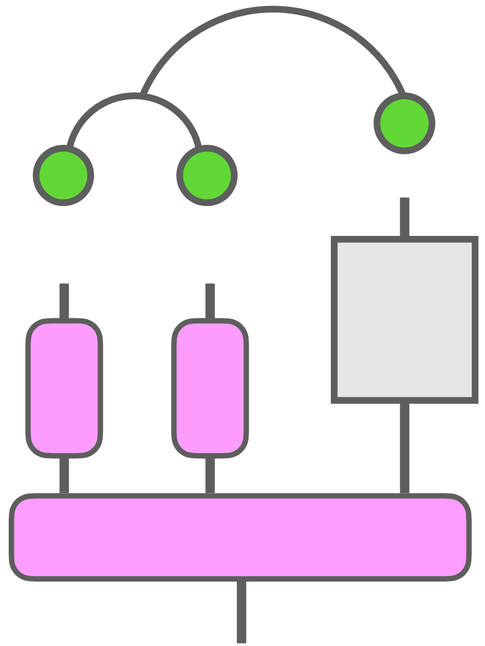
Durable Execution



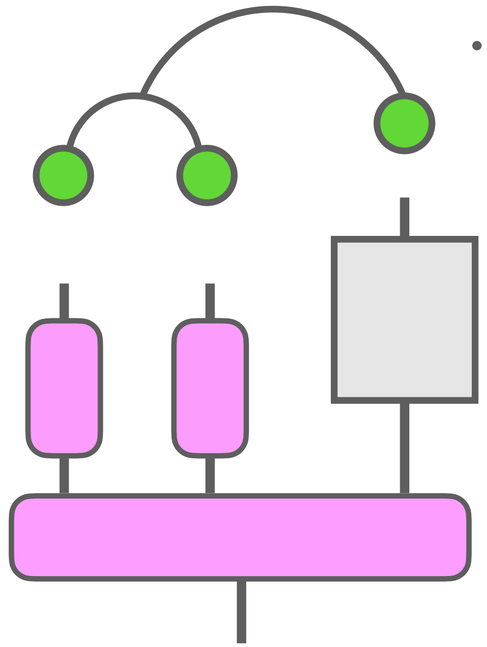
Durable Execution



Durable Execution



Durable Execution



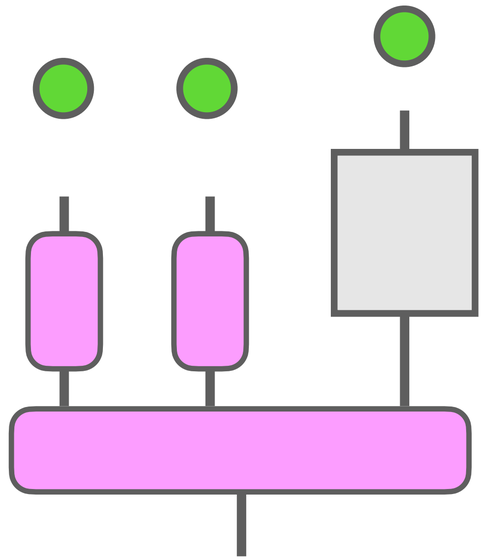
available inputs

- data
- persistable

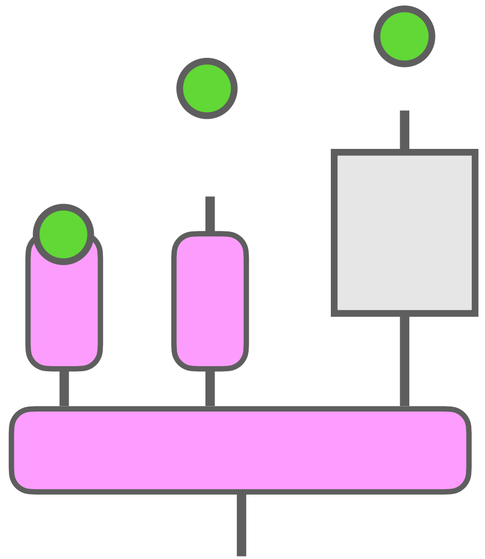
continuation

- data
- persistable

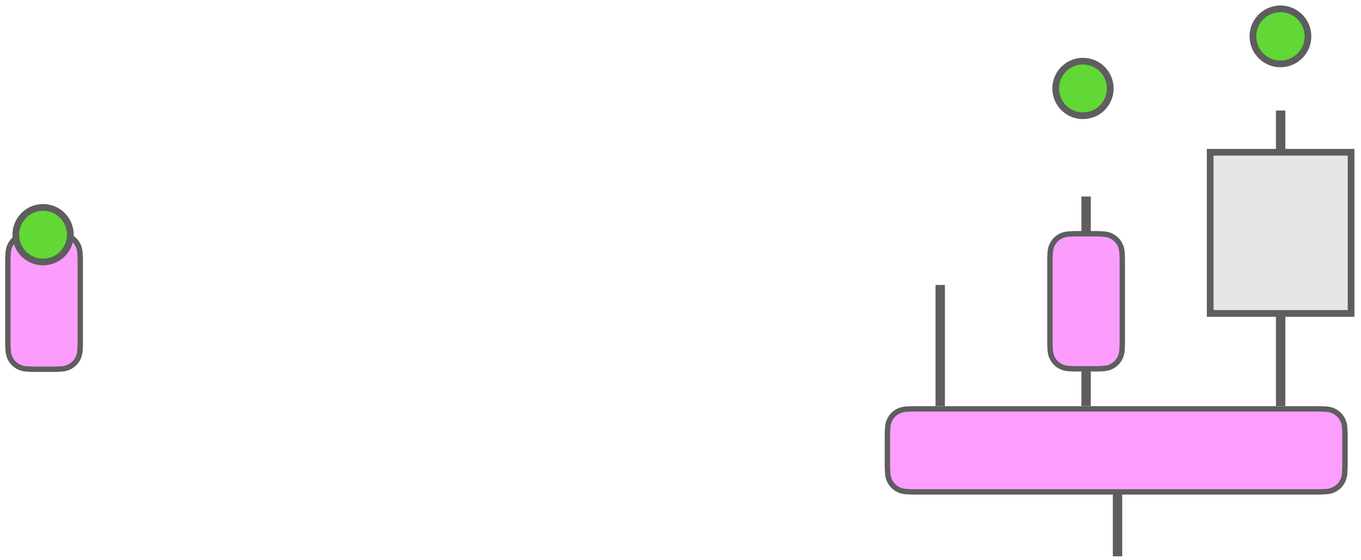
Durable Execution



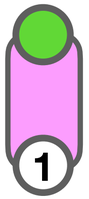
Durable Execution

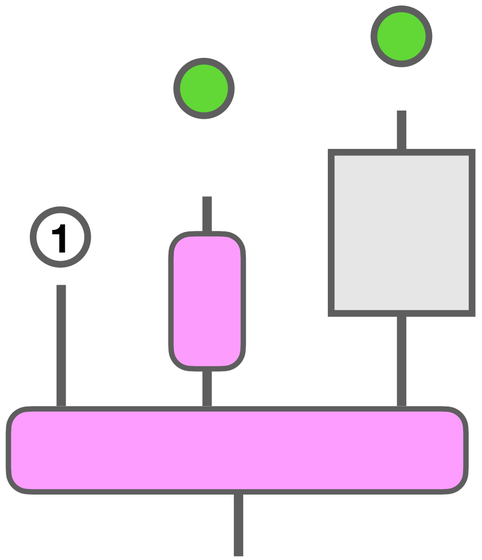


Durable Execution

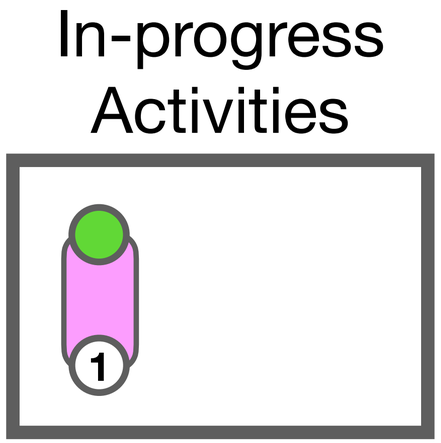


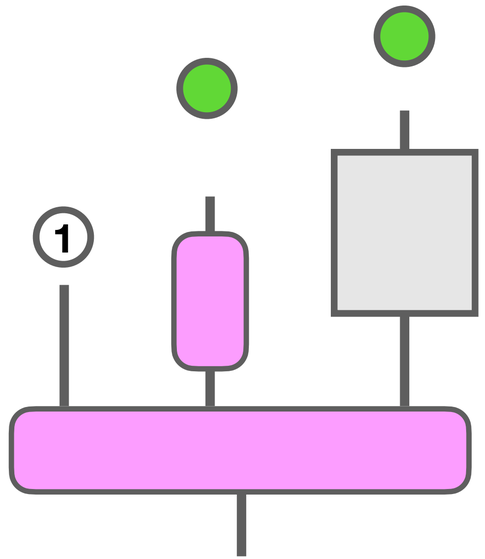
Durable Execution





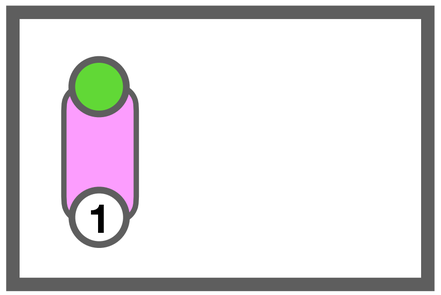
Durable Execution



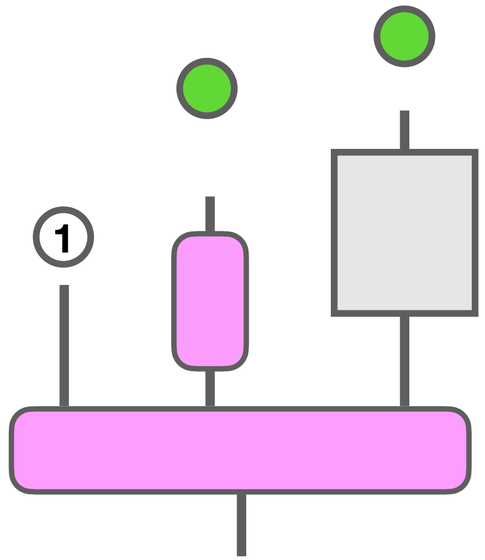


Durable Execution

In-progress Activities

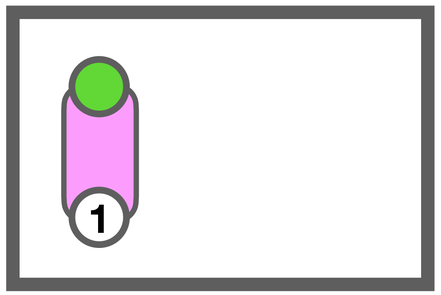


- persisted recipes
- restartable

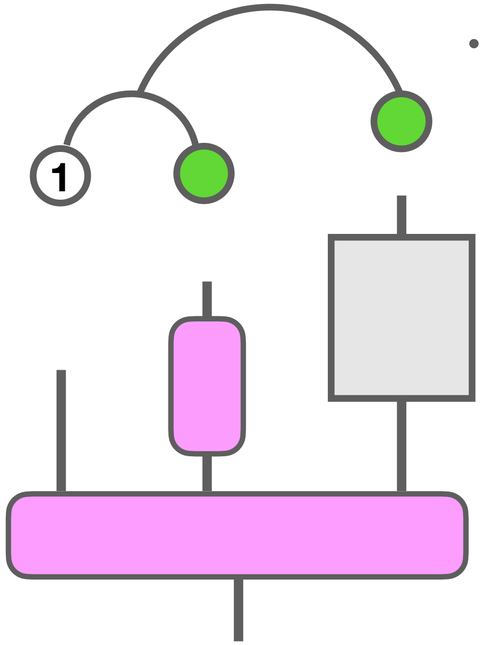


Durable Execution

In-progress Activities



- persisted recipes
- restartable



inputs

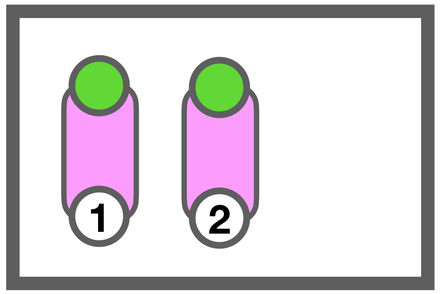
- available or *promised*
- persistable

continuation

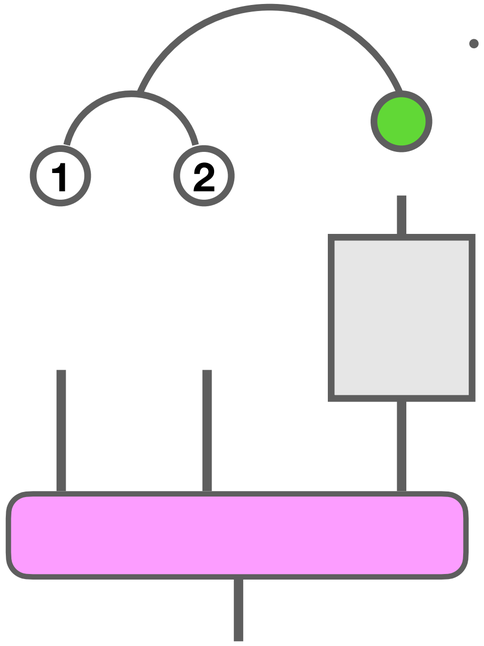
- data
- persistable

Durable Execution

In-progress Activities



- persisted recipes
- restartable



inputs

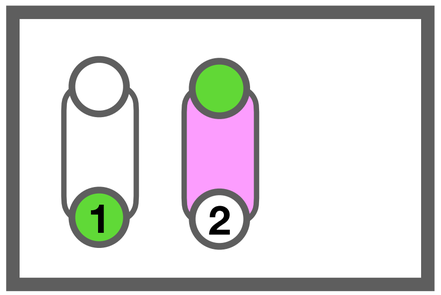
- available or *promised*
- persistable

continuation

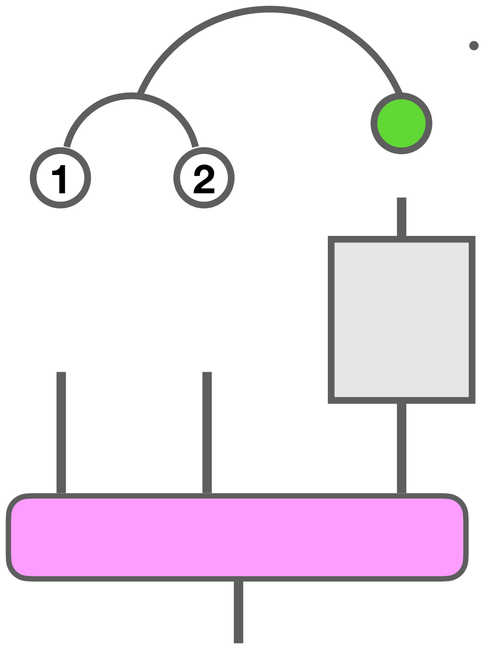
- data
- persistable

Durable Execution

In-progress Activities



- persisted recipes
- restartable



inputs

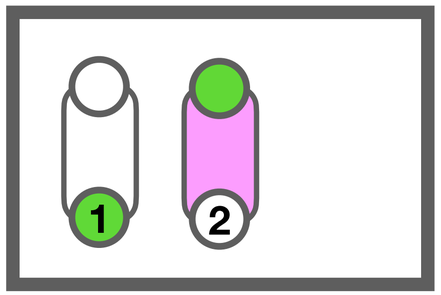
- available or *promised*
- persistable

continuation

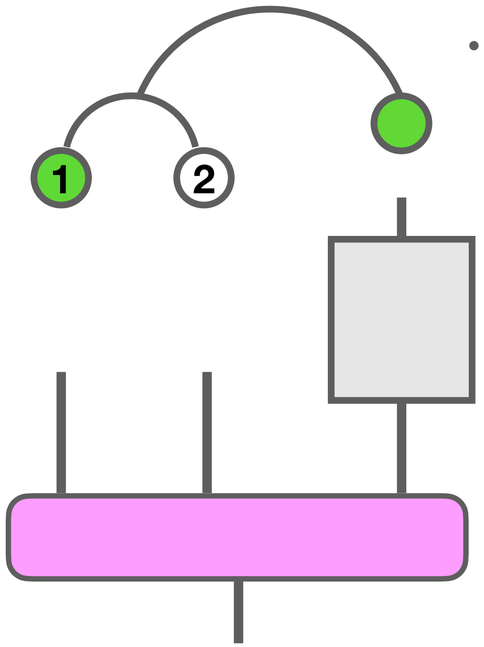
- data
- persistable

Durable Execution

In-progress Activities



- persisted recipes
- restartable



inputs

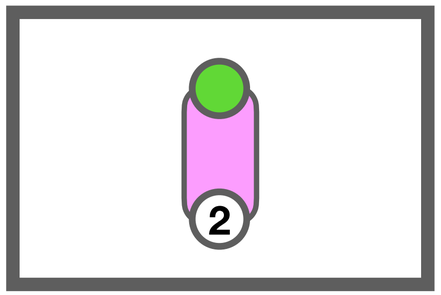
- available or *promised*
- persistable

continuation

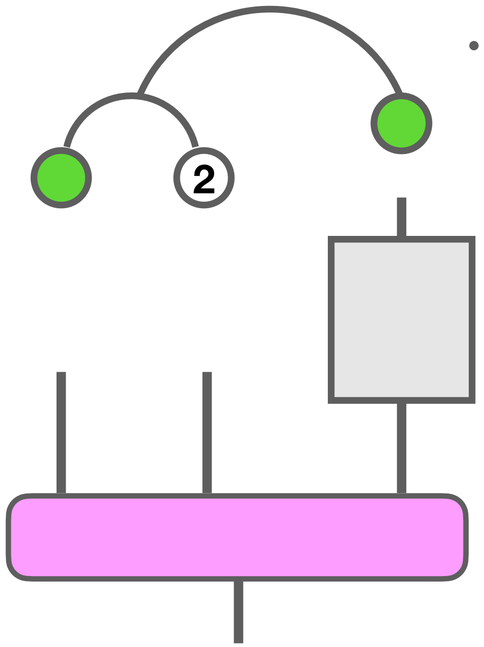
- data
- persistable

Durable Execution

In-progress Activities



- persisted recipes
- restartable



inputs

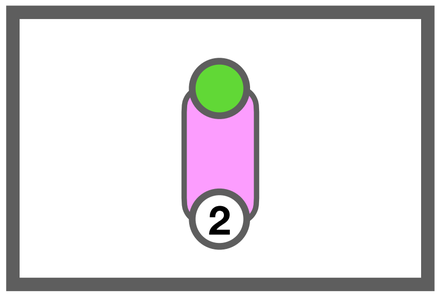
- available or *promised*
- persistable

continuation

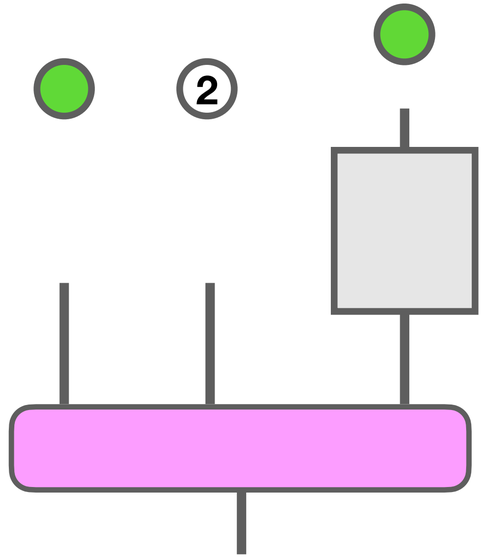
- data
- persistable

Durable Execution

In-progress Activities

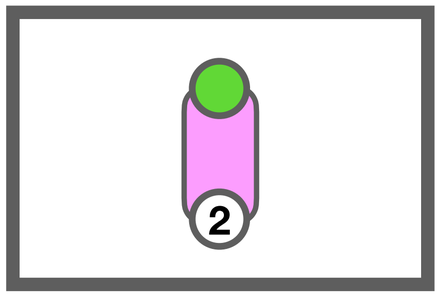


- persisted recipes
- restartable

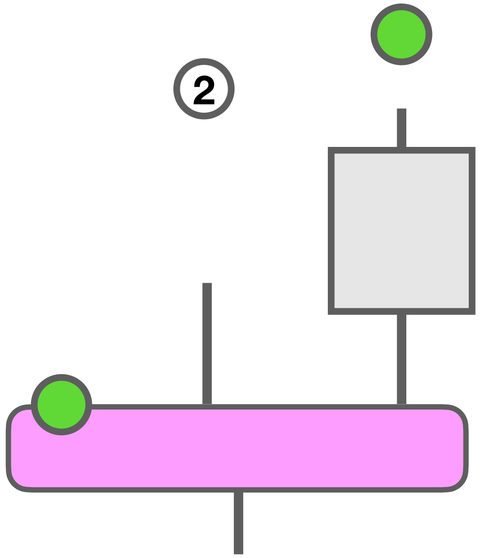


Durable Execution

In-progress Activities

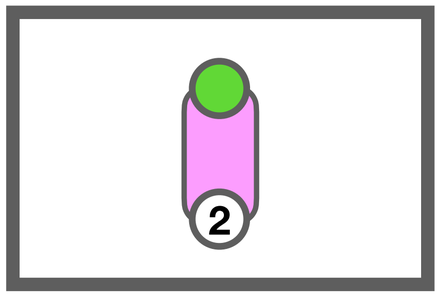


- persisted recipes
- restartable

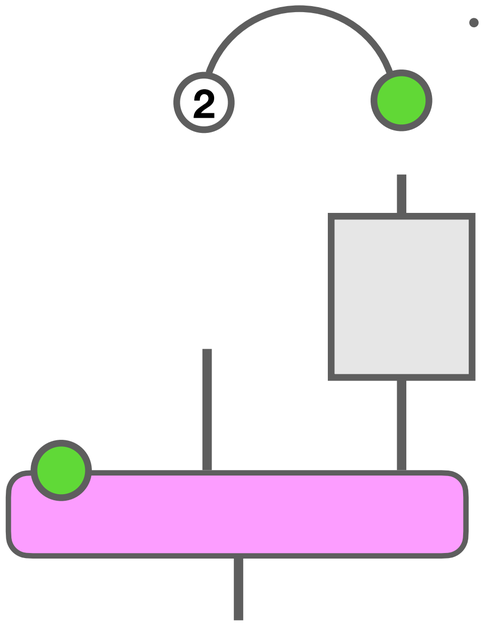


Durable Execution

In-progress Activities



- persisted recipes
- restartable



inputs

- available or *promised*
- persistable

continuation

- with *captured* values
- persistable

Durable Execution

Durable Execution

- **execution** state always **as data**

Durable Execution

- **execution** state always **as data**
- introspectable

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence
  - storage format

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence
  - storage format
    - graph?

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence
  - storage format
    - graph?
    - single document?

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence
  - storage format
    - graph?
    - single document?
- Orchestration

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence
  - storage format
    - graph?
    - single document?
- Orchestration
  - coordinator, workers, scheduler, …

Durable Execution

- **execution** state always **as data**
- introspectable
- opens possibilities
  - visualize
  - visualize mid-execution
  - edit mid-execution

Omitted “details”

- Serialization
  - pluggable for custom data types
- Persistence
  - storage format
    - graph?
    - single document?
- Orchestration
  - coordinator, workers, scheduler, …
- Externally completable promises

Distinctly Scala in Action

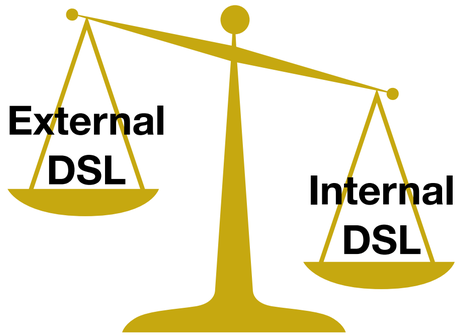
- Extension Methods `flow(expr) , expr switch {…}`
- Extractors `case id ** history =>`
- Context Functions

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

- Givens
- Opaque Types `Expr`
- Polymorphic Functions `sum: [X,Y] => (Flow[X, R], Flow[Y, R]) => Flow[X ++ Y, R]`
- lightweight Macros (source position, var names)
- Path-dependent types (not seen here, but heavily used in the library)

Take Aways

- **Programs-as-data** open new possibilities
- Does not take much to represent expressive control flows
- **Variables problematic**
  - Avoid internally to make some **illegal programs unrepresentable**
  - **Affordable** translation to point-free via canned implementation
- Expanding the case for internal DSLs



Thank you!

https://github.com/TomasMikula/libretto/tree/main/lambda-examples