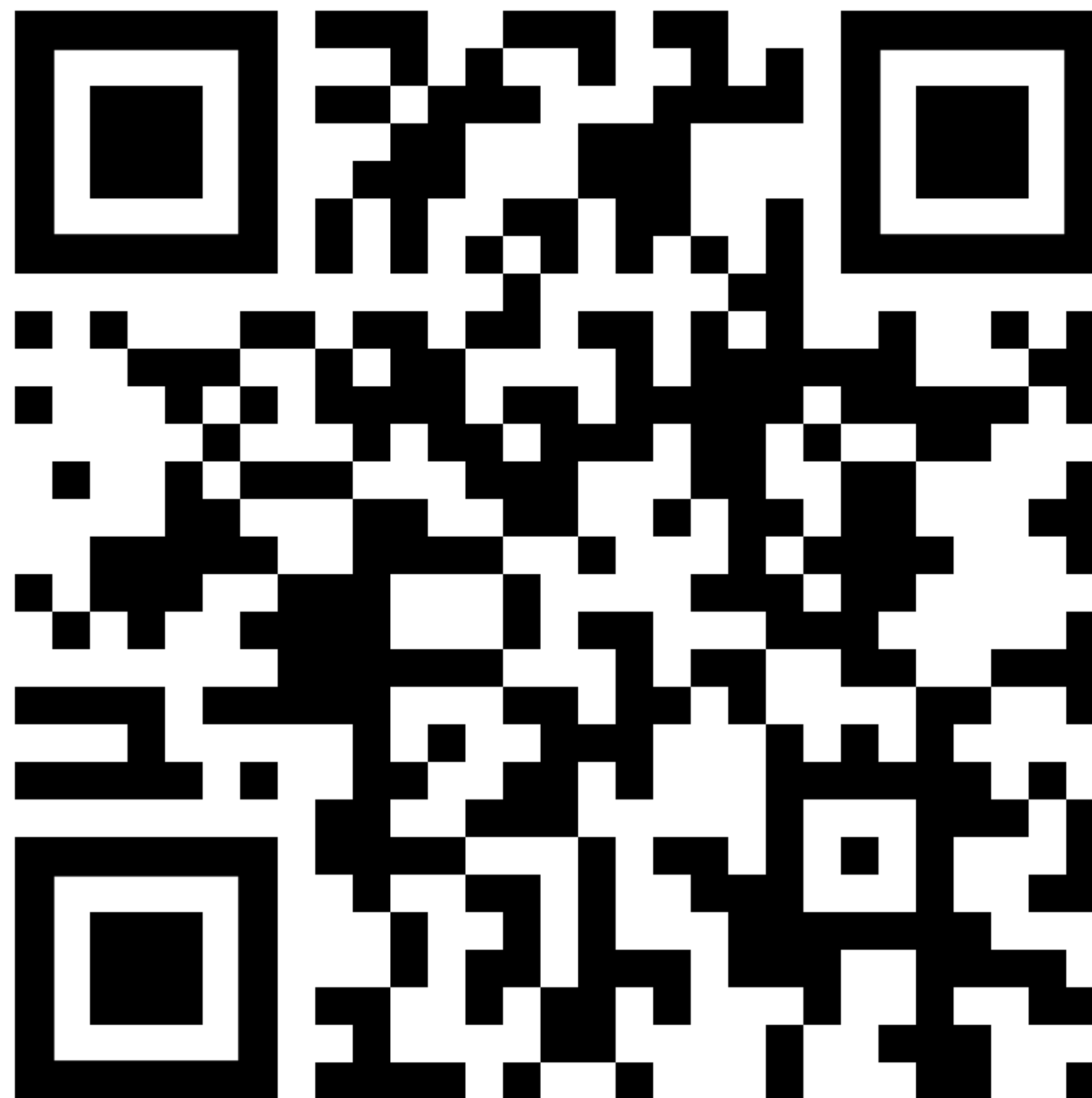


Типизированный язык выражений с помощью Dotty



<https://github.com/road21/talks>



Алексей Троицкий

@ Telegram, github: [road21](#)

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

registered_at <= account_opened_at

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

```
(count(account_opened_at == null) / count_all) < 0.20
```

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

Метрики

Число пользователей,
которые еще не открыли
счет
not_opened

Число всех пользователей
clients_count

Пример

Имя пользователя <i>name</i>	Дата регистрации <i>registered_at</i>	Дата открытия счета <i>account_opened_at</i>
Вася	2023-08-01	2023-08-01
Петя	2023-08-05	<i>null</i>
Вова	2023-08-05	2023-08-04

Метрики

Число пользователей,
которые еще не открыли
счет
not_opened

Число всех пользователей
clients_count

$(\text{not_opened} / \text{clients_count}) < 0.20$

Что там в DQ

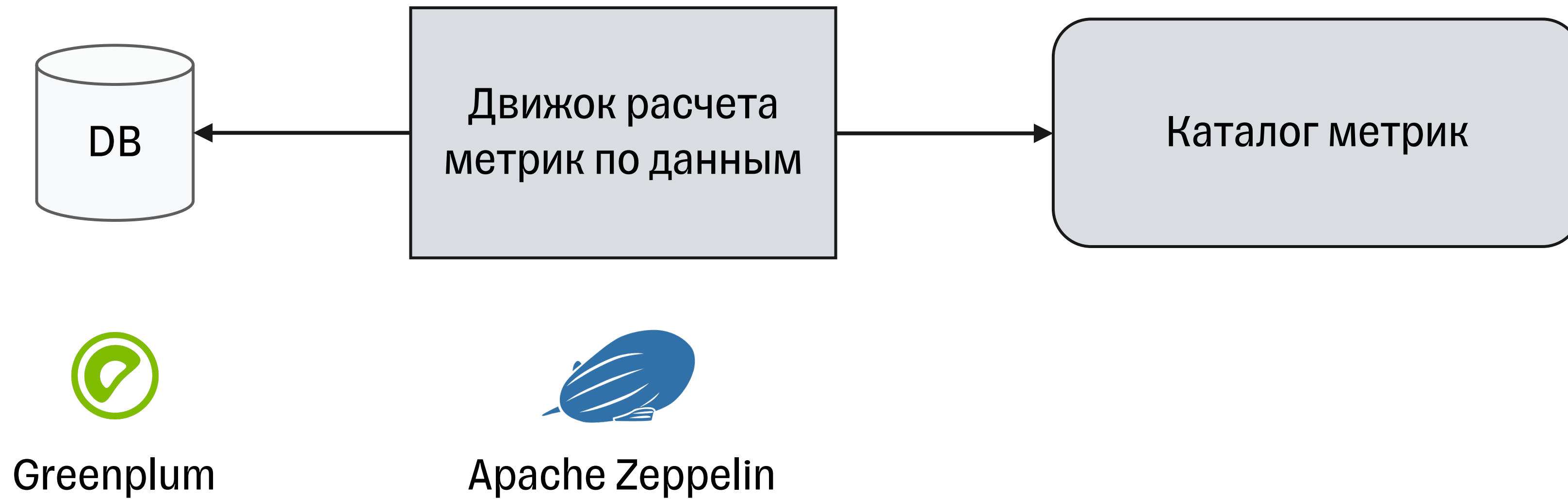


Что там в DQ

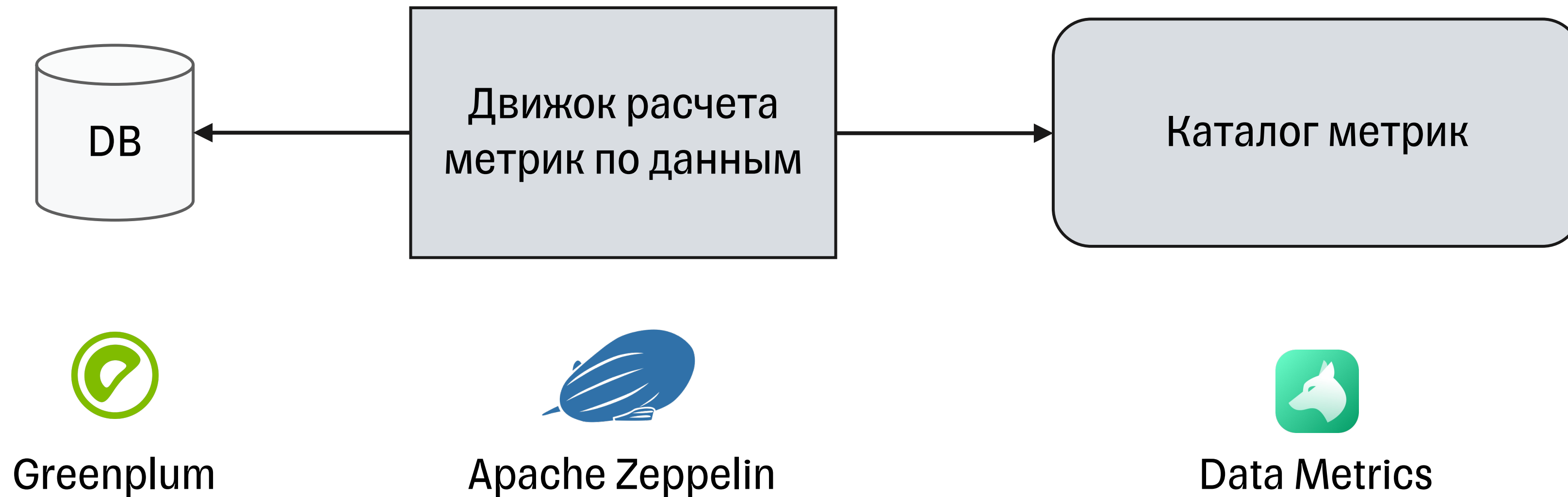


Greenplum

Что там в DQ



Что там в DQ



А язык выражений зачем?

Два типа метрик:

- «входные» метрики (значения присылаются из движка вычислений)
- «вычисляемые» метрики (значения вычисляются на основе других метрик)

А язык выражений зачем?

Два типа метрик:

- «входные» метрики (значения присылаются из движка вычислений)
- «вычисляемые» метрики (значения вычисляются на основе других метрик)

«Входные» метрики

Число пользователей,
которые еще не открыли
счет
not_opened

Число всех пользователей
clients_count

А язык выражений зачем?

Два типа метрик:

- «Входные» метрики (значения присылаются из движка вычислений)
- «Вычисляемые» метрики (значения вычисляются на основе других метрик)

«Входные» метрики

Число пользователей,
которые еще не открыли
счет

not_opened

Число всех пользователей
clients_count

«Вычисляемые» метрики

Относительное число
пользователей, которые еще не
открыли счет

**not_opened_rel = not_opened /
clients_count**

Постановка соглашений

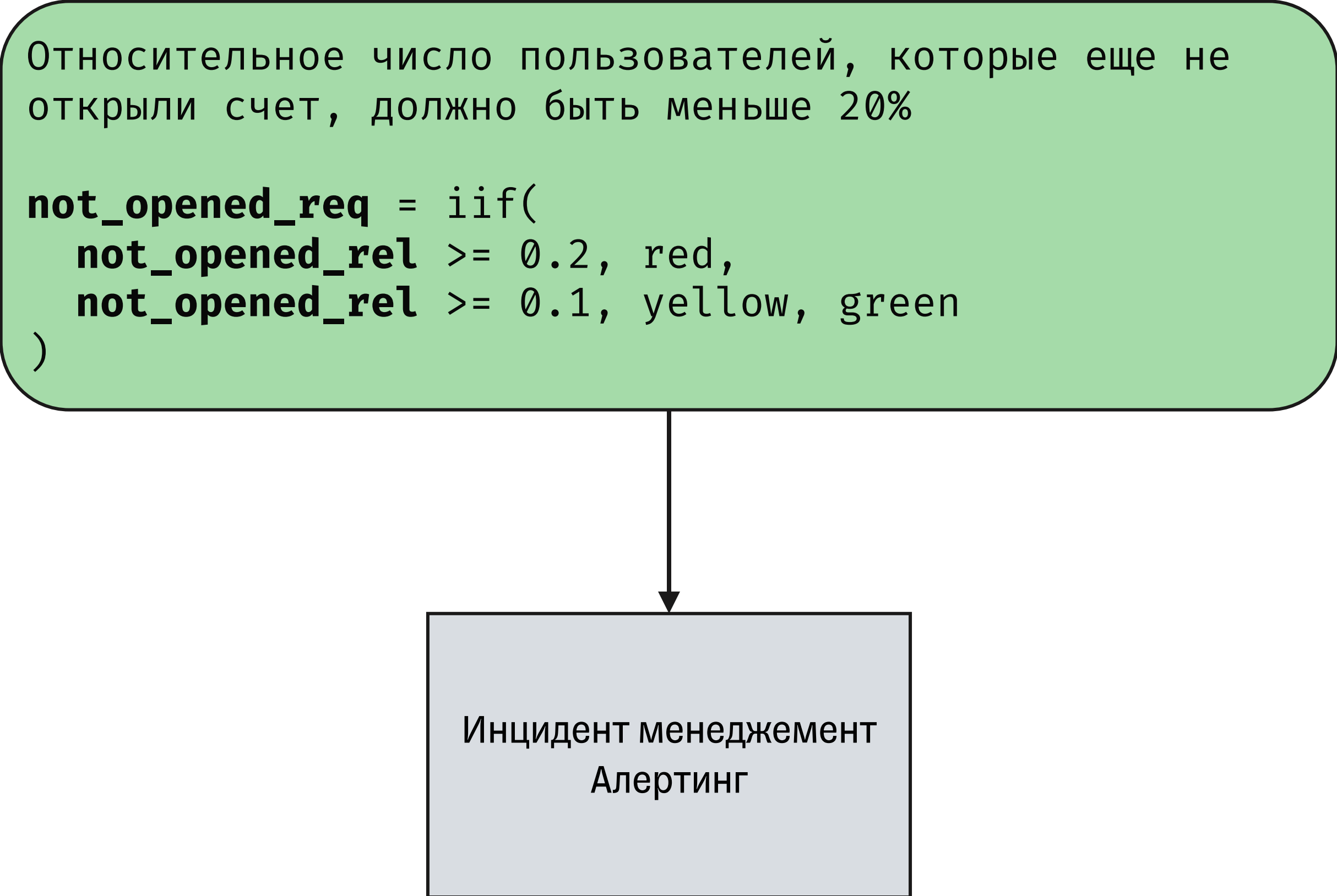
Относительное число пользователей, которые еще не открыли счет, должно быть меньше 20%

```
not_opened_req = iif(  
    not_opened_rel >= 0.2, red,  
    not_opened_rel >= 0.1, yellow, green  
)
```

Постановка соглашений

Относительное число пользователей, которые еще не открыли счет, должно быть меньше 20%

```
not_opened_req = iif(  
    not_opened_rel >= 0.2, red,  
    not_opened_rel >= 0.1, yellow, green  
)
```



Инцидент менеджмент
Алертинг

Типы метрик

- Decimal
- Boolean
- String
- RedYellowGreen
- Date
- DateTime

Типы метрик

- Decimal
- Boolean
- String
- RedYellowGreen
- Date
- DateTime

Число пользователей,
которые еще не открыли
счет

not_opened: Decimal

Число всех пользователей

clients_count: Decimal

Относительное число
пользователей, которые еще
не открыли счет

not_opened_rel: Decimal =
not_opened / clients_count

Относительное число пользователей, которые еще не
открыли счет, должно быть меньше 20%

```
not_opened_req: RedYellowGreen = iif(  
    not_opened_rel >= 0.2, red,  
    not_opened_rel >= 0.1, yellow, green  
)
```

Что нужно от языка

- Простой синтаксис
- Арифметические / строковые / булевы операции / работа с датой/временем
- Условный оператор
- Проверка типов

Что нужно от языка

- Простой синтаксис
- Арифметические / строковые / булевы операции / работа с датой/временем
- Условный оператор
- Проверка типов

- Под JVM
- Легко добавлять новые методы

ВЗЯТЬ ЧТО-ТО ГОТОВОЕ?

- Google CEL
<https://github.com/google/cel-java>
- Spring Expression Language
<https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html>
- EvalEx
<https://github.com/ezylang/EvalEx>

ВЗЯТЬ ЧТО-ТО ГОТОВОЕ?

- Google CEL
<https://github.com/google/cel-java>
- Spring Expression Language
<https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html>
- EvalEx
<https://github.com/ezylang/EvalEx>



Написать самому?

```
enum Tree:
```

```
  case Ident(name: String)
```

```
  case If(cond: Tree, `then`: Tree, `else`: Tree)
```

```
  case InfixOp(left: Tree, op: Op, right: Tree)
```

```
  case Literal(x: Constant)
```

```
enum Op:
```

```
  case `+`, `-`, `/`, `*`, `and`, `or`, `>=`, `<=`, `>`, `<`, `==`
```

```
enum Constant:
```

```
  case Bool(v: Boolean)
```

```
  case Str(v: String)
```

```
  case Decimal(v: Double)
```

```
  case RYG(v: RedYellowGreen)
```

Написать самому?

enum Tree:

```
case Ident(name: String)
case If(cond: Tree, `then`: Tree, `else`: Tree)
case InfixOp(left: Tree, op: Op, right: Tree)
case Literal(x: Constant)
```

enum Op:

```
case `+`, `-`, `/`, `*`, `and`
```

enum Constant:

```
case Bool(v: Boolean)
case Str(v: String)
case Decimal(v: Double)
case RYG(v: RedYellowGreen)
```

```
iif(
  not_opened_rel >= 0.2, red,
  not_opened_rel >= 0.1, yellow, green
)
```

```
If(
  InfixOp(Ident("not_opened_rel"), Op.`>=`,
    Literal(Decimal(0.2))),
  Literal(RYG(red)),
  If(
    InfixOp(Ident("not_opened_rel"), Op.`>=`,
      Literal(Decimal(0.1))),
    Literal(RYG(yellow)),
    Literal(RYG(green))
  )
)
```

Написать самому?

```
def parser(input: String): Either[ParseError, Tree] = ???
```

```
def typer(ast: Tree, metrics: Map[MetricName, MetricType]):  
    Either[TypeError, Unit] = ???
```

```
def eval(ast: Tree, metrics: Map[MetricName, MetricValue]):  
    Either[RuntimeError, MetricValue] = ???
```

Написать самому?

```
def parser(input: String): Either[ParseError, Tree] = ???
```

```
def typer(ast: Tree, metrics: Map[MetricName, MetricType]):  
    Either[TypeError, Unit] = ???
```

```
def eval(ast: Tree, metrics: Map[MetricName, MetricValue]):  
    Either[RuntimeError, MetricValue] = ???
```

Написать самому?

```
def parser(input: String): Either[ParseError, Tree] = ???
```

```
def typer(ast: Tree, metrics: Map[MetricName, MetricType]):  
    Either[TypeError, Unit] = ???
```

```
def eval(ast: Tree, metrics: Map[MetricName, MetricValue]):  
    Either[RuntimeError, MetricValue] = ???
```

Написать самому?

```
def parser(input: String): Either[ParseError, Tree] = ???
```

```
def typer(ast: Tree, metrics: Map[MetricName, MetricType]):  
  Either[TypeError, Unit] = ???
```

```
def eval(ast: Tree, metrics: Map[MetricName, MetricValue]):  
  Either[RuntimeError, MetricValue] = ???
```

GADT

```
enum Tree[A <: Type]:  
  case Ident[A <: Type](ref: Ref[A])  
    extends Tree[A]  
  
  case If[A <: Type](cond: Tree[Boolean], `then`: Tree[A], `else`: Tree[A])  
    extends Tree[A]  
  
  case NumOp(left: Tree[Decimal], op: NumOp, right: Tree[Decimal])  
    extends Tree[Decimal]  
  
  case BoolOp(left: Tree[Boolean], op: BoolOp, right: Tree[Boolean])  
    extends Tree[Boolean]  
  
  case Literal[A <: Type](c: Constant[A])  
    extends Tree[A]
```

GADT

```
enum Tree[A <: Type]:  
  case Ident[A <: Type](ref: Ref[A])  
    extends Tree[A]  
  
  case If[A <: Type](cond: Tree[Bool], `then`: Tree[A], `else`: Tree[A])  
    extends Tree[A]  
  
  case NumOp(left: Tree[Decimal], op: NumOp, right: Tree[Decimal])  
    extends Tree[Decimal]  
  
  case BoolOp(left: Tree[Bool], op: BoolOp, right: Tree[Bool])  
    extends Tree[Bool]  
  
  case Literal[A <: Type](c: Constant[A])  
    extends Tree[A]
```


GADT

```
enum Tree[A <: Type]:  
  case Ident[A <: Type](ref: Ref[A])  
    extends Tree[A]  
  
  case If[A <: Type](cond: Tree[Bool], `then`: Tree[A], `else`: Tree[A])  
    extends Tree[A]  
  
  case NumOp(left: Tree[Decimal], op: NumOp, right: Tree[Decimal])  
    extends Tree[Decimal]  
  
  case BoolOp(left: Tree[Bool], op: BoolOp, right: Tree[Bool])  
    extends Tree[Bool]  
  
  case Literal[A <: Type](c: Constant[A])  
    extends Tree[A]
```

GADT

```
enum Tree[A <: Type]:  
  case Ident[A <: Type](ref: Ref[A])  
    extends Tree[A]
```

```
  case If[A <: Type](cond: Tree[Bool], `then`: Tree[A], `else`: Tree[A])  
    extends Tree[A]
```

```
  case NumOp(left: Tree[Decimal], op: NumOp, right: Tree[Decimal])  
    extends Tree[Decimal]
```

```
  case BoolOp(left: Tree[Bool], op: BoolOp, right: Tree[Bool])  
    extends Tree[Bool]
```

```
  case Literal[A <: Type](c: Constant[A])  
    extends Tree[A]
```

Typed Tagless Final Interpreters
Oleg Kiselyov

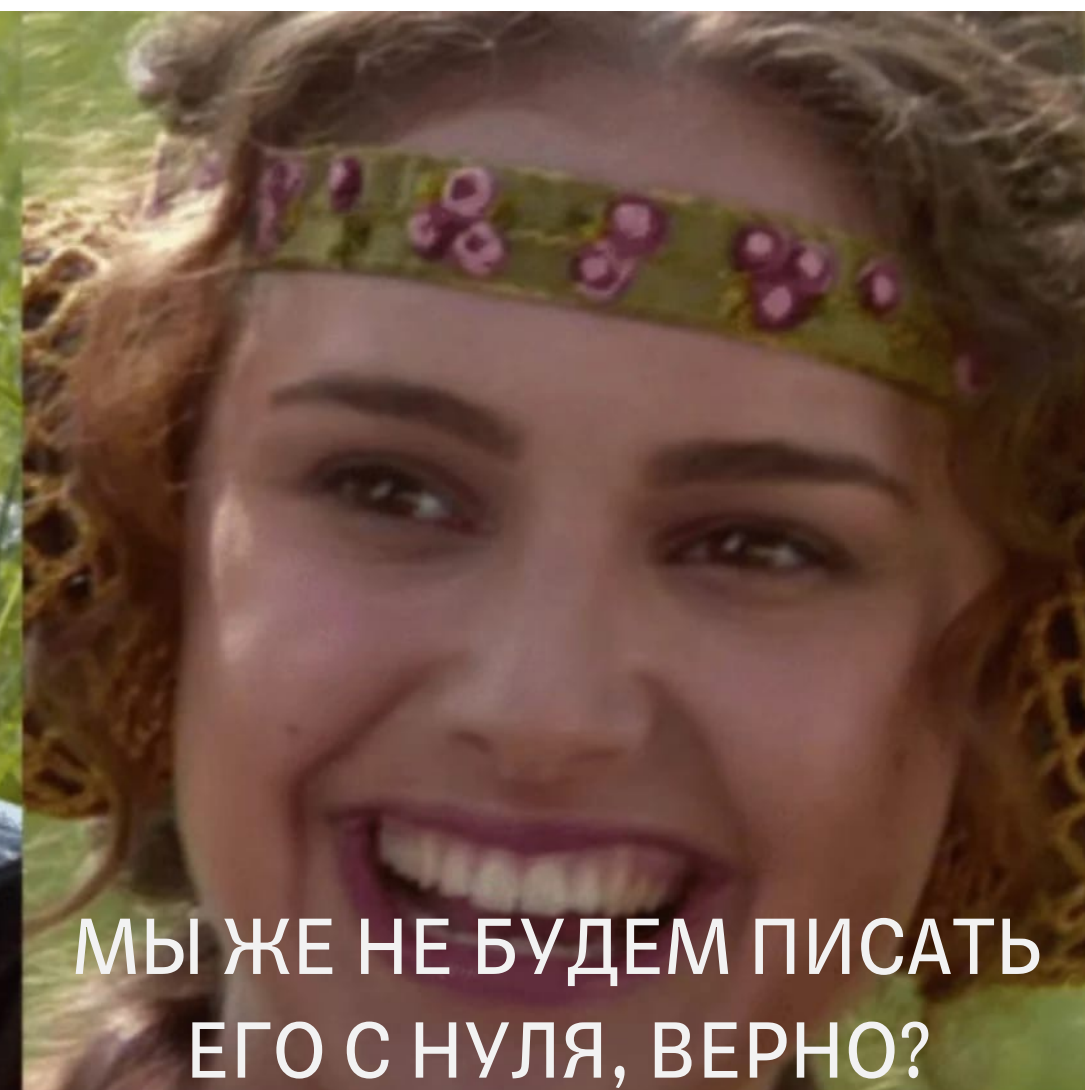
<https://okmij.org/ftp/tagless-final/#course-oxford>

Вопросы

- Как расширять?
- Насколько читабельно?

Вопросы

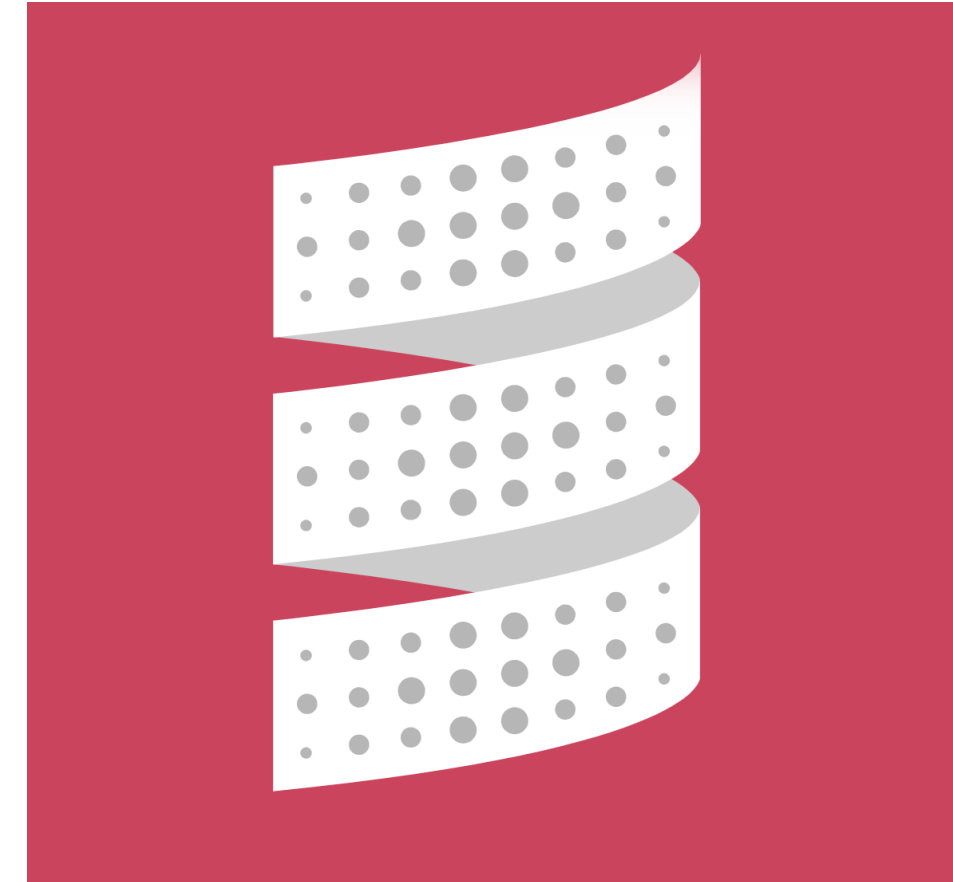
- Как расширять?
- Насколько читабельно?



**Но ведь это уже все
реализовано...**

Dotty

- Компилятор scala 3+
- Написан на scala 3
- Разрабатывается в EPFL
- В основе система типов DOT-calculus (*dependent object types*)



Compiler

```
class Compiler {  
  def phases: List[List[Phase]] =  
    frontendPhases ::: picklerPhases ::: transformPhases ::: backendPhases  
  /** Phases dealing with the frontend up to trees ready for TASTY pickling */  
  protected def frontendPhases: List[List[Phase]] =  
    List(new Parser) :: // Compiler frontend: scanner, parser  
    List(new TyperPhase) :: // Compiler frontend: namer, typer  
    ...  
  
  /** Generate the output of the compilation */  
  protected def backendPhases: List[List[Phase]] =  
    List(new backend.sjs.GenSJSIR) :: // Generate .sjsir files for Scala.js (not  
enabled by default)  
    List(new GenBCode) :: // Generate JVM bytecode  
    Nil
```

Compiler

```
class Compiler {  
  def phases: List[List[Phase]] =  
    frontendPhases ::: picklerPhases ::: transformPhases ::: backendPhases  
  /** Phases dealing with the frontend up to trees ready for TASTY pickling */  
  protected def frontendPhases: List[List[Phase]] =  
    List(new Parser) :: // Compiler frontend: scanner, parser  
    List(new TyperPhase) :: // Compiler frontend: namer, typer  
    ...  
  
  /** Generate the output of the compilation */  
  protected def backendPhases: List[List[Phase]] =  
    List(new backend.sjs.GenSJSIR) :: // Generate .sjsir files for Scala.js (not  
enabled by default)  
    List(new GenBCode) :: // Generate JVM bytecode  
    Nil
```


Compiler

```
class Compiler {  
  def phases: List[List[Phase]] =  
    frontendPhases ::: picklerPhases ::: transformPhases ::: backendPhases  
  /** Phases dealing with the frontend up to trees ready for TASTY pickling */  
  protected def frontendPhases: List[List[Phase]] =  
    List(new Parser) :: // Compiler frontend: scanner, parser  
    List(new TyperPhase) :: // Compiler frontend: namer, typer  
    ...  
  
  /** Generate the output of the compilation */  
  protected def backendPhases: List[List[Phase]] =  
    List(new backend.sjs.GenSJSIR) :: // Generate .sjsir files for Scala.js (not  
enabled by default)  
    List(new GenBCode) :: // Generate JVM bytecode  
    Nil
```

Compiler

```
class Compiler {  
  def phases: List[List[Phase]] =  
    frontendPhases ::: picklerPhases ::: transformPhases ::: backendPhases  
  /** Phases dealing with the frontend up to trees ready for TASTY pickling */  
  protected def frontendPhases: List[List[Phase]] =  
    List(new Parser) :: // Compiler frontend: scanner, parser  
    List(new TyperPhase) :: // Compiler frontend: namer, typer  
    ...  
  
  /** Generate the output of the compilation */  
  protected def backendPhases: List[List[Phase]] =  
    List(new backend.sjs.GenSJSIR) :: // Generate .sjsir files for Scala.js (not  
enabled by default)  
    List(new GenBCode) :: // Generate JVM bytecode  
    Nil
```

Tree

```
type Untyped = Type | Null

abstract class Tree[+T <: Untyped](
  implicit @constructorOnly src: SourceFile
) { ... }
```

Tree

```
case class Ident[+T <: Untyped] private[ast] (  
  name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }  
  
case class Literal[+T <: Untyped] private[ast] (  
  const: Constant  
) (implicit @constructorOnly src: SourceFile) extends Tree[T] with TermTree[T] { ... }  
  
case class If[+T <: Untyped] private[ast] (  
  cond: Tree[T], thenp: Tree[T], elsep: Tree[T]  
) (implicit @constructorOnly src: SourceFile) extends TermTree[T] { ... }  
  
case class Apply[+T <: Untyped] private[ast] (  
  fun: Tree[T], args: List[Tree[T]]  
) (implicit @constructorOnly src: SourceFile) ...  
  
case class Select[+T <: Untyped] private[ast] (  
  qualifier: Tree[T], name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```

Tree

```
case class Ident[+T <: Untyped] private[ast] (  
  name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```



```
case class Literal[+T <: Untyped] private[ast] (  
  const: Constant  
) (implicit @constructorOnly src: SourceFile) extends Tree[T] with TermTree[T] { ... }
```

```
case class If[+T <: Untyped] private[ast] (  
  cond: Tree[T], thenp: Tree[T], elsep: Tree[T]  
) (implicit @constructorOnly src: SourceFile) extends TermTree[T] { ... }
```

```
case class Apply[+T <: Untyped] private[ast] (  
  fun: Tree[T], args: List[Tree[T]]  
) (implicit @constructorOnly src: SourceFile) ...
```

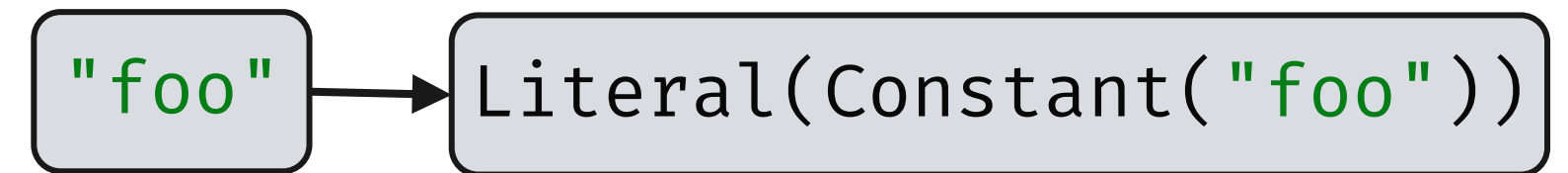
```
case class Select[+T <: Untyped] private[ast] (  
  qualifier: Tree[T], name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```

Tree

```
case class Ident[+T <: Untyped] private[ast] (  
  name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```



```
case class Literal[+T <: Untyped] private[ast] (  
  const: Constant  
) (implicit @constructorOnly src: SourceFile) extends Tree[T] with TermTree[T] { ... }
```



```
case class If[+T <: Untyped] private[ast] (  
  cond: Tree[T], thenp: Tree[T], elsep: Tree[T]  
) (implicit @constructorOnly src: SourceFile) extends TermTree[T] { ... }
```

```
case class Apply[+T <: Untyped] private[ast] (  
  fun: Tree[T], args: List[Tree[T]]  
) (implicit @constructorOnly src: SourceFile) ...
```

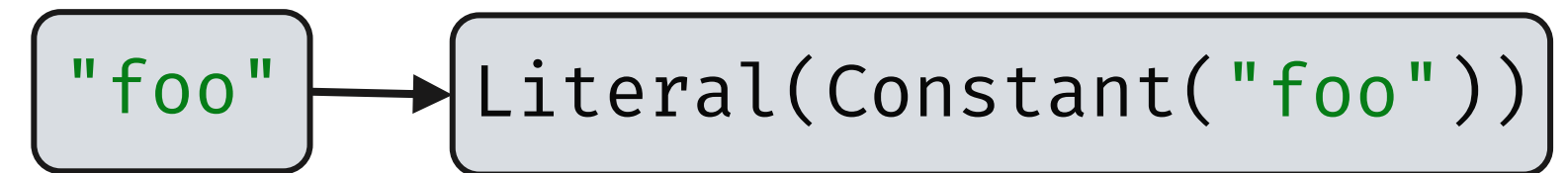
```
case class Select[+T <: Untyped] private[ast] (  
  qualifier: Tree[T], name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```

Tree

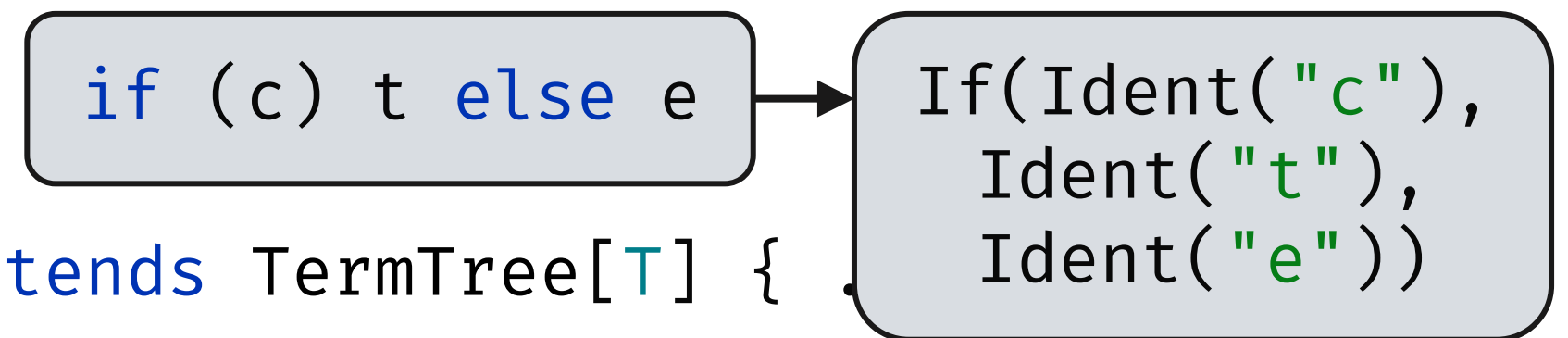
```
case class Ident[+T <: Untyped] private[ast] (  
  name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```



```
case class Literal[+T <: Untyped] private[ast] (  
  const: Constant  
) (implicit @constructorOnly src: SourceFile) extends Tree[T] with TermTree[T] { ... }
```



```
case class If[+T <: Untyped] private[ast] (  
  cond: Tree[T], thenp: Tree[T], elsep: Tree[T]  
) (implicit @constructorOnly src: SourceFile) extends TermTree[T] { ... }
```



```
case class Apply[+T <: Untyped] private[ast] (  
  fun: Tree[T], args: List[Tree[T]]  
) (implicit @constructorOnly src: SourceFile) ...
```

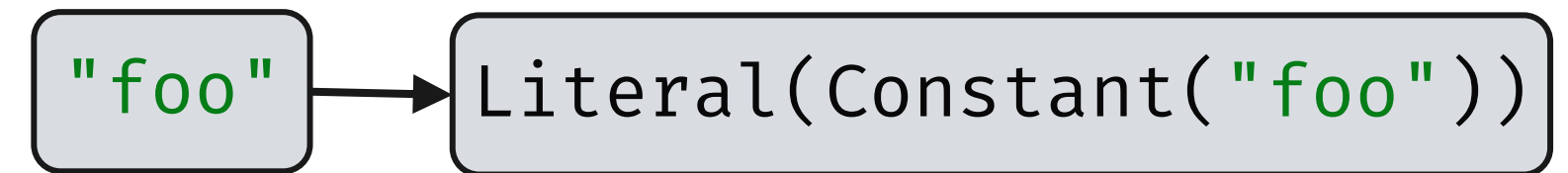
```
case class Select[+T <: Untyped] private[ast] (  
  qualifier: Tree[T], name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```


Tree

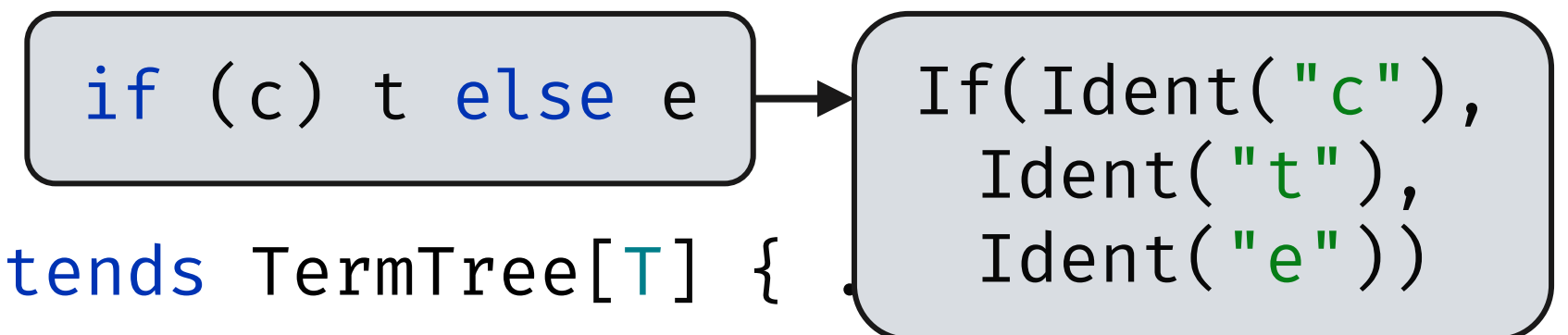
```
case class Ident[+T <: Untyped] private[ast] (  
  name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```



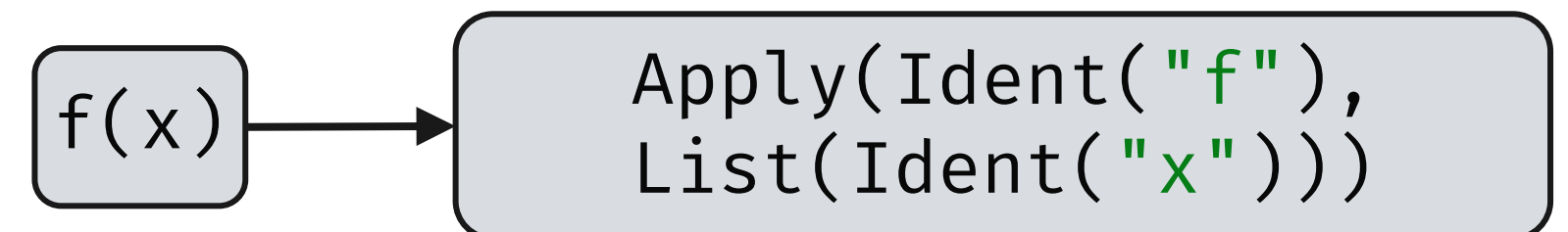
```
case class Literal[+T <: Untyped] private[ast] (  
  const: Constant  
) (implicit @constructorOnly src: SourceFile) extends Tree[T] with TermTree[T] { ... }
```



```
case class If[+T <: Untyped] private[ast] (  
  cond: Tree[T], thenp: Tree[T], elsep: Tree[T]  
) (implicit @constructorOnly src: SourceFile) extends TermTree[T] { ... }
```



```
case class Apply[+T <: Untyped] private[ast] (  
  fun: Tree[T], args: List[Tree[T]]  
) (implicit @constructorOnly src: SourceFile) ...
```



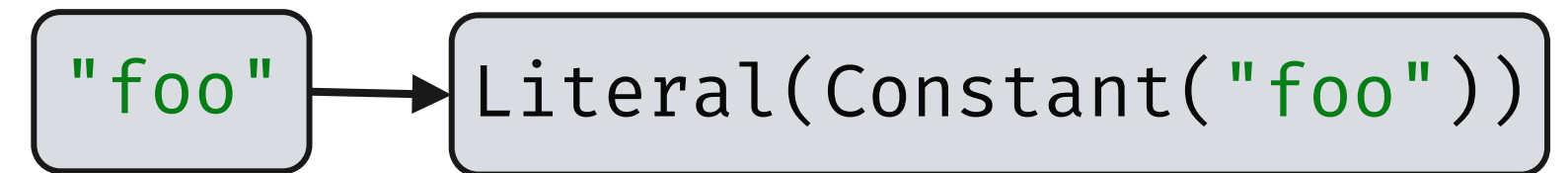
```
case class Select[+T <: Untyped] private[ast] (  
  qualifier: Tree[T], name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```


Tree

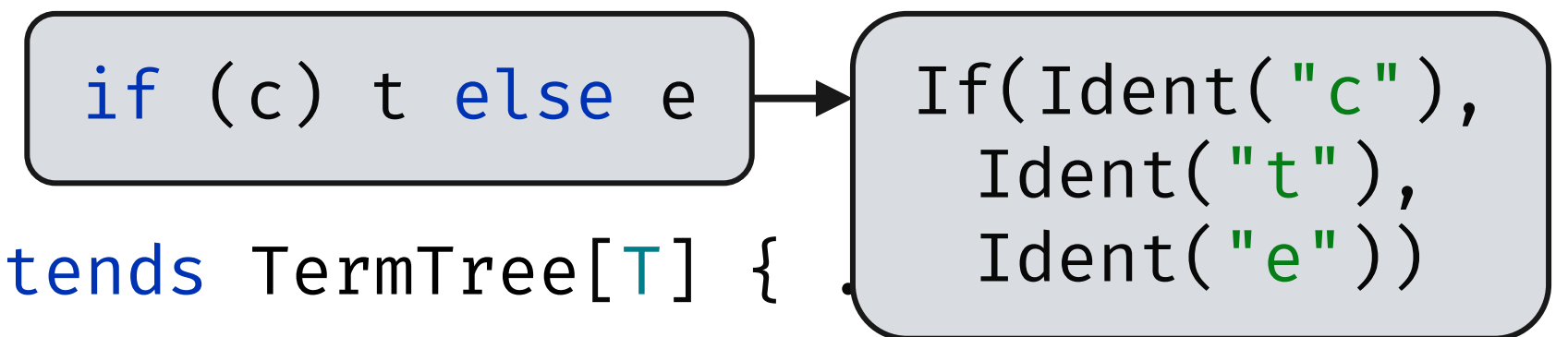
```
case class Ident[+T <: Untyped] private[ast] (  
  name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```



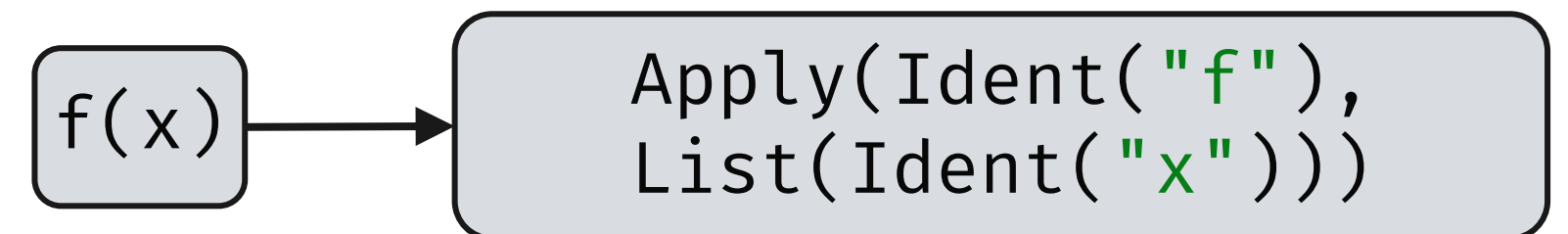
```
case class Literal[+T <: Untyped] private[ast] (  
  const: Constant  
) (implicit @constructorOnly src: SourceFile) extends Tree[T] with TermTree[T] { ... }
```



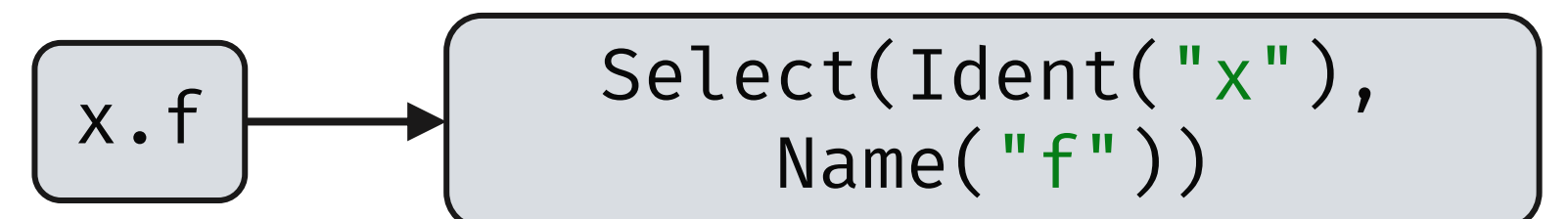
```
case class If[+T <: Untyped] private[ast] (  
  cond: Tree[T], thenp: Tree[T], elsep: Tree[T]  
) (implicit @constructorOnly src: SourceFile) extends TermTree[T] { ... }
```



```
case class Apply[+T <: Untyped] private[ast] (  
  fun: Tree[T], args: List[Tree[T]]  
) (implicit @constructorOnly src: SourceFile) ...
```

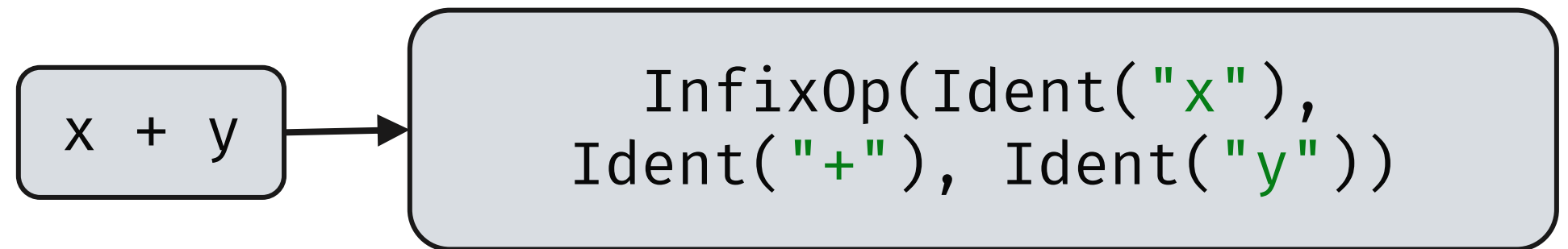


```
case class Select[+T <: Untyped] private[ast] (  
  qualifier: Tree[T], name: Name  
) (implicit @constructorOnly src: SourceFile) extends RefTree[T] { ... }
```



Tree

```
case class InfixOp(  
  left: Tree, op: Ident, right: Tree  
)(  
  implicit @constructorOnly src: SourceFile  
) extends OpTree // <: Tree[Untyped]
```



Tree

```
if (y != 0)
  result(x / y)
else error
```

Tree

```
if (y != 0)
  result(x / y)
else error
```

dotty.tools.untpd.Tree

```
If(
  InfixOp(
    Ident("y"),
    Ident("!="),
    Literal(Constant(0))
  ),
  Apply(
    Ident("result"),
    List(
      InfixOp(
        Ident("x"),
        Ident("/"),
        Ident("y")
      )
    )
  ),
  Ident("error")
)
```

Tree

```
if (y != 0)
  result(x / y)
else error
```

dotty.tools.untpd.Tree

```
If(
  InfixOp(
    Ident("y"),
    Ident("!="),
    Literal(Constant(0))
  ),
  Apply(
    Ident("result"),
    List(
      InfixOp(
        Ident("x"),
        Ident("/"),
        Ident("y")
      )
    )
  ),
  Ident("error")
)
```

dotty.tools.tpd.Tree

```
If(
  Apply(
    Select(
      Ident("y"), Name("!=")
    ),
    Constant(0)
  ),
  Apply(
    Ident("result"),
    List(
      Apply(
        Select(
          Ident("x"),
          Name("/")
        ), Ident("y")
      )
    )
  ),
  Ident("error")
)
```

Tree

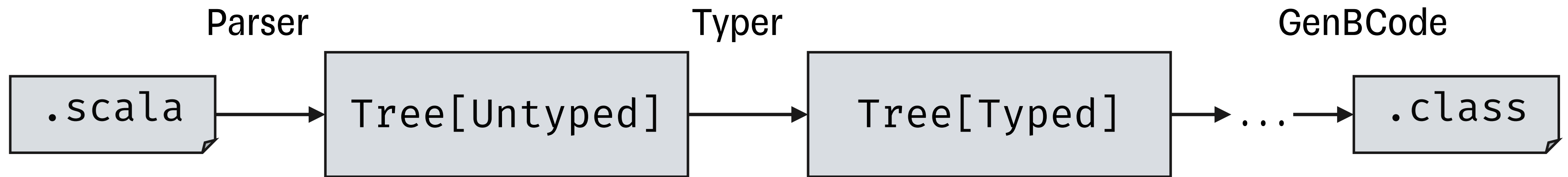
```
if (y != 0)
  result(x / y)
else error
```

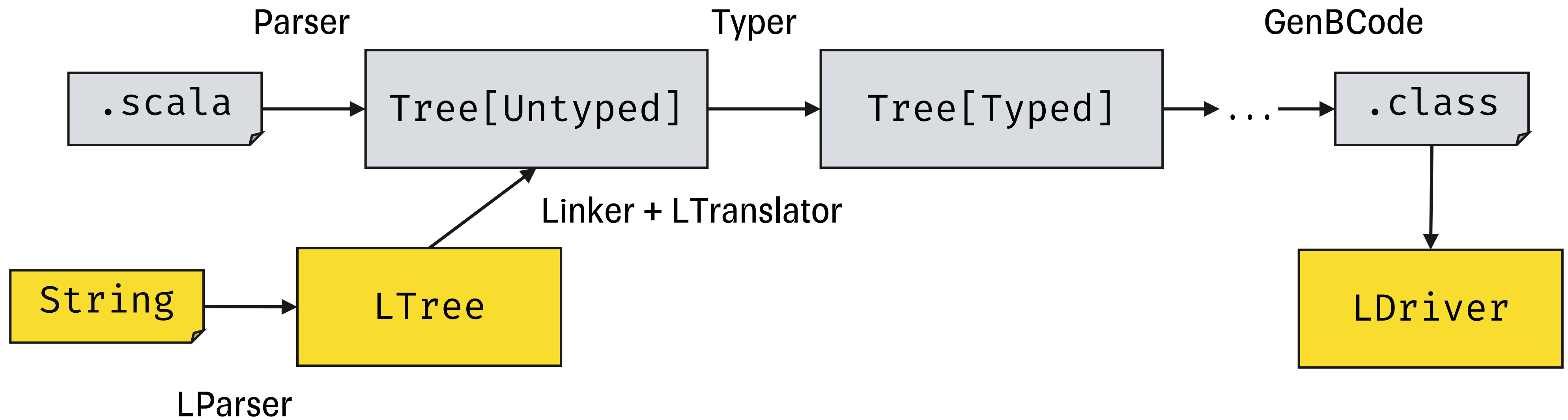
dotty.tools.untpd.Tree

```
If(
  InfixOp(
    Ident("y"),
    Ident("!="),
    Literal(Constant(0))
  ),
  Apply(
    Ident("result"),
    List(
      InfixOp(
        Ident("x"),
        Ident("/"),
        Ident("y")
      )
    )
  ),
  Ident("error")
)
```

dotty.tools.tpd.Tree

```
If(
  Apply(
    Select(
      Ident("y"), Name("!=")
    ),
    Constant(0)
  ),
  Apply(
    Ident("result"),
    List(
      Apply(
        Select(
          Ident("x"),
          Name("/")
        ), Ident("y")
      )
    )
  ),
  Ident("error")
)
```





Подключаем Dotty

```
libraryDependencies += Seq(  
  "org.scala-lang" %% "scala3-compiler" % Version.scalaLib,  
  "org.scala-lang" %% "scala3-library"   % Version.scalaLib,  
)
```

Переопределяем Compiler

```
class LCompiler(term: LTree, outputType: LType) extends Compiler:  
  override protected def frontendPhases: List[List[Phase]] =  
    List(new LTranslator(term, outputType)) ::  
      List(new TyperPhase) :: ...
```

Переопределяем Compiler

```
class LCompiler(term: LTree, outputType: LType) extends Compiler:  
  override protected def frontendPhases: List[List[Phase]] =  
    List(new LTranslator(term, outputType)) ::  
      List(new TyperPhase) :: ...
```

Трансляция

```
def run(args: Arguments): Any =  
  iif(  
    ref[BigDecimal]("not_opened_rel", args) >= 0.2, red,  
    ref[BigDecimal]("not_opened_rel", args) >= 0.1, yellow, green  
  )
```

```
iif(  
  not_opened_rel >= 0.2, red,  
  not_opened_rel >= 0.1, yellow, green  
)
```

Трансляция

```
def run(args: Arguments): Any =
```

```
  iif(
```

```
    ref[BigDecimal]("not_opened_rel", args) >= 0.2, red,
```

```
    ref[BigDecimal]("not_opened_rel", args) >= 0.1, yellow, green
```

```
  )
```

```
//-----
```

```
def iif[A <: Out, B <: Out, C <: Out, Out](
```

```
  cond: Boolean, `then`: => A, cond2: => Boolean, then2: => B, `else`: => C
```

```
): Out = ...
```

```
case class Arguments(calc: Map[String, Any])
```

```
def ref[Type](name: String, args: Arguments): Type =
```

```
  args.calc(name).asInstanceOf[Type]
```

```
iif(
  not_opened_rel >= 0.2, red,
  not_opened_rel >= 0.1, yellow, green
)
```

Трансляция

```
def run(args: Arguments): Any =
```

```
  iif(
```

```
    ref[BigDecimal]("not_opened_rel", args) >= 0.2, red,
```

```
    ref[BigDecimal]("not_opened_rel", args) >= 0.1, yellow, green
```

```
  )
```

```
//-----
```

```
def iif[A <: Out, B <: Out, C <: Out, Out](
```

```
  cond: Boolean, `then`: => A, cond2: => Boolean, then2: => B, `else`: => C
```

```
): Out = ...
```

```
case class Arguments(calc: Map[String, Any])
```

```
def ref[Type](name: String, args: Arguments): Type =
```

```
  args.calc(name).asInstanceOf[Type]
```

```
iif(
  not_opened_rel >= 0.2, red,
  not_opened_rel >= 0.1, yellow, green
)
```

LTree

```
enum LTree:  
  case Literal(p: Constant)  
  case Apply(l: LTree, r: Vector[LTree])  
  case Ident(name: Identifier)  
  case InfixOp(left: LTree, op: Ident, right: LTree)  
  case Select(qualifier: LTree, name: Identifier)  
  case MetricRef(id: RefId, calcType: LType)
```

LTree

```
enum LTree:
  case Literal(p: Constant)
  case Apply(l: LTree, r: Vector[LTree])
  case Ident(name: Identifier)
  case InfixOp(left: LTree, op: Ident, right: LTree)
  case Select(qualifier: LTree, name: Identifier)
  case MetricRef(id: RefId, calcType: LType)
```

```
iif(
  not_opened_rel >= 0.2, red,
  not_opened_rel >= 0.1, yellow, green
)
```

LParse

```
Apply(
  Ident("iif"),
  Vector(
    InfixOp(
      Ident("not_opened_rel"),
      Ident(">="),
      Literal(Constant.Decimal(0.2))
    ),
    Ident("red"),
    InfixOp(
      Ident("not_opened_rel"),
      Ident(">="),
      Literal(Constant.Decimal(0.1))
    ),
    Ident("yellow"),
    Ident("green")
  )
)
```


LTree

```
enum LTree:
  case Literal(p: Constant)
  case Apply(l: LTree, r: Vector[LTree])
  case Ident(name: Identifier)
  case InfixOp(left: LTree, op: Ident, right: LTree)
  case Select(qualifier: LTree, name: Identifier)
```

MetricRef("not_opened_rel", LType.Decimal)

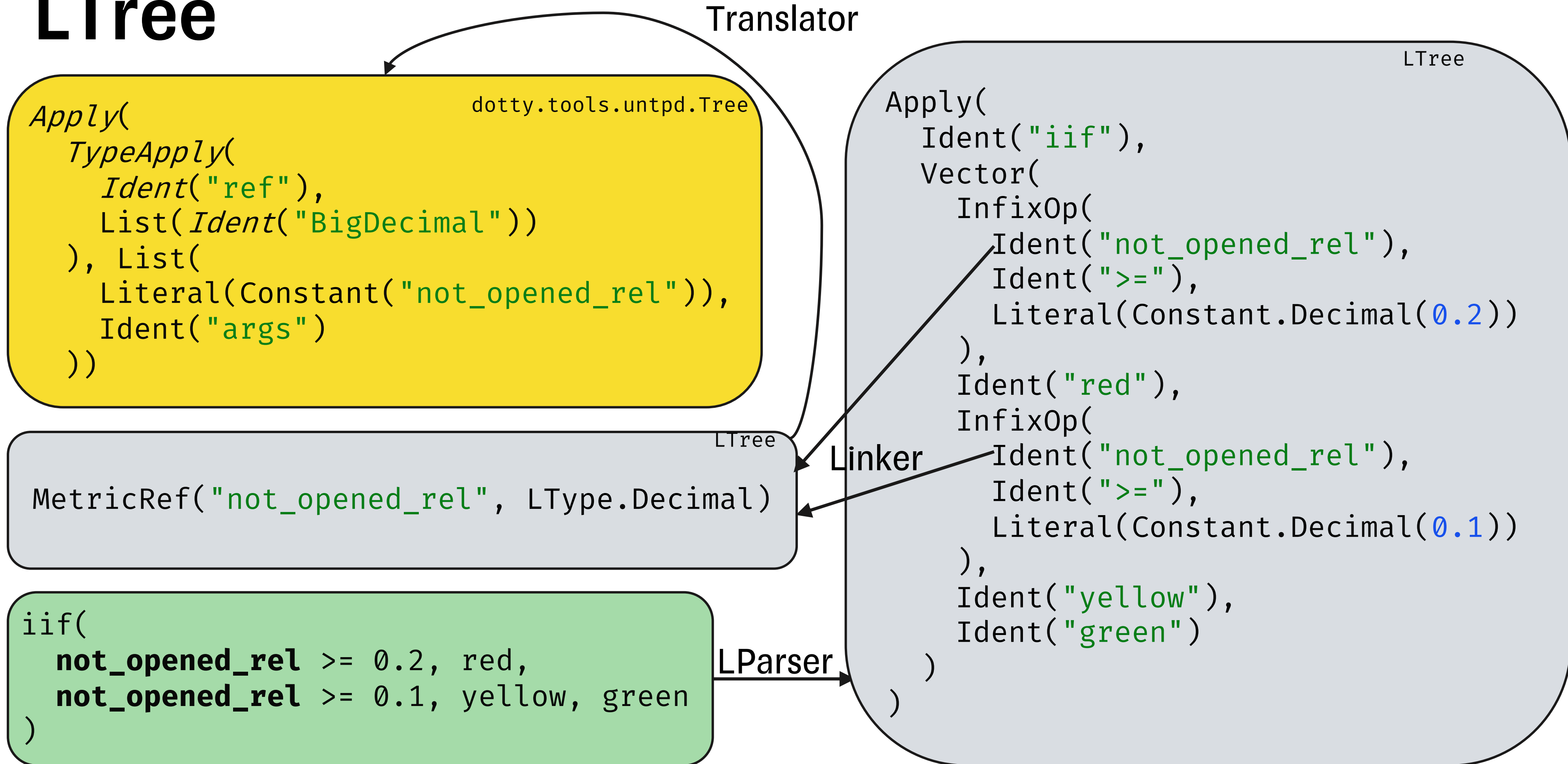
```
iif(
  not_opened_rel >= 0.2, red,
  not_opened_rel >= 0.1, yellow, green
)
```

```
Apply(
  Ident("iif"),
  Vector(
    InfixOp(
      Ident("not_opened_rel"),
      Ident(">="),
      Literal(Constant.Decimal(0.2))
    ),
    Ident("red"),
    InfixOp(
      Ident("not_opened_rel"),
      Ident(">="),
      Literal(Constant.Decimal(0.1))
    ),
    Ident("yellow"),
    Ident("green")
  )
)
```

Linker

LPParser

LTree



Что делаем с байткодом

- При создании метрик компилируем выражения и сохраняем байткод в базу
- При запуске расчета метрик читаем байткод из базы и запускаем через рефлексия

Что делаем с байткодом

- При создании метрик компилируем выражения и сохраняем байткод в базу
- При запуске расчета метрик читаем байткод из базы и запускаем через рефлексия

```
Try {  
  val cl      = URLClassLoader(Array(dir.toUri.toURL),  
this.getClass.getClassLoader)  
  val cls     = cl.loadClass(calcCls)  
  val method = cls.getMethod("run", arguments.getClass)  
  method.invoke(null, arguments).asInstanceOf[outputType.Repr]  
}.recoverWith { case ex: InvocationTargetException =>  
  Failure(ex.getTargetException)  
}.toEither.left.map(errors.Runtime.apply)
```

Что делаем с байткодом

- При создании метрик компилируем выражения и сохраняем байткод в базу
- При запуске расчета метрик читаем байткод из базы и запускаем через рефлексия

```
Try {  
  val cl      = URLClassLoader(Array(dir.toUri.toURL),  
this.getClass.getClassLoader)  
  val cls     = cl.loadClass(calcCls)  
  val method = cls.getMethod("run", arguments.getClass)  
  method.invoke(null, arguments).asInstanceOf[outputType.Repr]  
}.recoverWith { case ex: InvocationTargetException =>  
  Failure(ex.getTargetException)  
}.toEither.left.map(errors.Runtime.apply)
```

Что делаем с байткодом

- При создании метрик компилируем выражения и сохраняем байткод в базу
- При запуске расчета метрик читаем байткод из базы и запускаем через рефлексия

```
Try {  
  val cl      = URLClassLoader(Array(dir.toUri.toURL),  
this.getClass.getClassLoader)  
  val cls     = cl.loadClass(calcCls)  
  val method = cls.getMethod("run", arguments.getClass)  
  method.invoke(null, arguments).asInstanceOf[outputType.Repr]  
}.recoverWith { case ex: InvocationTargetException =>  
  Failure(ex.getTargetException)  
}.toEither.left.map(errors.Runtime.apply)
```

Ошибки

- Ошибки парсинга
- Ошибки компиляции
- Ошибки при запуске расчета метрики

Ошибки

- Ошибки парсинга
- Ошибки компиляции
 - NotFound
 - TypeMismatch
- Ошибки при запуске расчета метрики

Плюсы

- Есть проверка типов!
- Из коробки стандартная библиотека `scala`
- Расширение = написание `scala` методов в «стандартном» пакете
- ~1к строк кода

Минусы

- Насколько стабилен интерфейс Dotty?
- Скорость компиляции
- Работа с ошибками

Бонус

```
account_opened_at_min >= toDate("2023-08-05")
```

Бонус

```
account_opened_at_min >= toDate("2023-08- 05")
```

Бонус

```
account_opened_at_min >= toDate("2023-08- 05")
```

```
inline def toDate(inline str: String): LocalDate =  
  ${ DateMacro.toDateCode('str) }
```

```
object DateMacro:  
  def toDateCode(str: Expr[String])(using Quotes): Expr[LocalDate] =  
    str.value match  
      case Some(value) =>  
        Try(LocalDate.parse(value)).fold(_ => report.errorAndAbort("Expected date,  
got: " + value, str), Expr.apply)  
      case None => '{ LocalDate.parse($str) }
```

Бонус

```
account_opened_at_min >= toDate("2023-08- 05")
```

```
inline def toDate(inline str: String): LocalDate =  
  ${ DateMacro.toDateCode('str) }
```

```
object DateMacro:  
  def toDateCode(str: Expr[String])(using Quotes): Expr[LocalDate] =  
    str.value match  
      case Some(value) =>  
        Try(LocalDate.parse(value)).fold(_ => report.errorAndAbort("Expected date,  
got: " + value, str), Expr.apply)  
      case None => '{ LocalDate.parse($str) }
```

Бонус

```
account_opened_at_min >= toDate("2023-08- 05")
```

```
inline def toDate(inline str: String): LocalDate =  
  ${ DateMacro.toDateCode('str) }
```

```
object DateMacro:  
  def toDateCode(str: Expr[String])(using Quotes): Expr[LocalDate] =  
    str.value match  
      case Some(value) =>  
        Try(LocalDate.parse(value)).fold(_ => report.errorAndAbort("Expected date,  
got: " + value, str), Expr.apply)  
      case None => '{ LocalDate.parse($str) }
```

Бонус

```
account_opened_at_min >= toDate("2023-08- 05")
```

```
inline def toDate(inline str: String): LocalDate =  
  ${ DateMacro.toDateCode('str) }
```

```
object DateMacro:
```

```
  def toDateCode(str: Expr[String])  
    str.value match  
      case Some(value) =>  
        Try(LocalDate.parse(value)).fold(_ => report.errorAndAbort("Expected date,  
got: " + value, str), Expr.apply)  
      case None => '{ LocalDate.parse($str) }
```

```
[error] -- Error  
[error] |      toDate("2023-08- 05")  
[error] |                ^^^^^^^^^^^^^^^  
[error] |      Expected date, got: 2023-08- 05
```




Спасибо!

Поделись своим впечатлением о докладе

