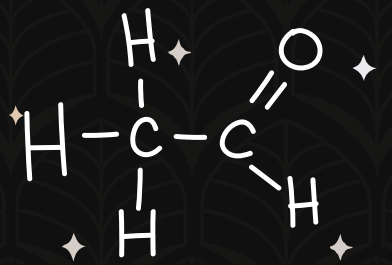
THE STRANGE, WACKY WORLD OF QUOTED-DSLS

Alexander Ioffe
@deusaquilus
ZIVERGE
quill

Code Samples Here: https://github.com/deusaquilus/qdsls-presentation

Caveat Emptor...

- All examples are in Scala 3
- Certain implementation details glossed
- Refer to linked examples for specifics



01

Why you should care

Once Upon a Time:

We had a model

```
case class RemoteData(
  def symbol: String
  def strikePrice: Int
  def underlyingPrice: Int
)
```

...and some data

```
RemoteData(AAPL, 140, 161)   // 21/AAPL
RemoteData(MSFT, 220, 337)   // 117/MSFT
RemoteData(NFLX, 600, 654)   // 54/NFLX
RemoteData(GOOG, 2800, 2936) // 136/GOOG
...
// 20 Billion More...
```

...and a query

```
def execute(data: RemoteData) =
  toString(data.underlyingPrice - data.strikePrice) + "/" + data.symbol
```

Once Upon a Time:

...and a JAR file

```
> sbt package -Denv=remote
...
[info] compiling 3942 Scala sources to /some/path/that/goes/on/forever
...
[success] Total time: 39 hours, completed Unidecember 01, 2302 12:00:00 AM

> scp target/job.jar emr-host:/home/hadoop/lib # and wait 4 hours
```

...and a cluster

```
> ssh -i /someplace/with/my/key.pem hadoop@emr-host
...
> spark-submit \
   --conf spark.sql.memory=SomethingInsaneGBs \
   --conf spark.yarn.driver.memoryOverhead=SomethingElseInsaneGBs \
   --conf spark.executor.instances=SizeOfMongolArmy \
   --conf 100-more-config-params \
   job.jar
```

...and life was hard!

```
> spark submit ... job.jar
2020/11/12 15:09:50 App report for application_0013 (state: RUNNING)
2021/10/23 15:09:53 App report for application_0013 (state: RUNNING)
...
2042/01/01 15:09:54 App report for application_0013 (state: STILL RUNNING)
...
Exception in thread "main" org.apache.spark.SparkException: Application
application_0013 finished with failed status
    at org.apache.spark.deploy.yarn.Client.run(Client.scala:1034)
    at org.apache.spark.deploy.yarn.Client$.main(Client.scala:1081)
```

```
CREATE TABLE
   parquet_remote_data (
   symbol STRING,
   stirkePrice INT,
   underlyingPrice Int
) STORED AS PARQUET;


SELECT
  toString(
    data.underlyingPrice -
    data.strikePrice
  ) + '/'
  data.symbol
FROM Data data
```

```
query("""SELECT
  CASE WHEN
    (data.underlyingPrice - data.strikePrice) *
    multiplier > someArbitraryAmount
  THEN
    toString(
      ((data.underlyingPrice - data.strikePrice) *
        multiplier) - initialAmount
    ) + '/' + sanitize(data.symbol)
  ELSE
    toString(
      (someUdf(data.underlyingPrice) -
        data.strikePrice)
    ) + '/' + sanitize(data.symbol) + otherQualifer
  data.symbol
FROM Data data
WHERE data.someField >= someAribtraryValue
""")
```

Language Embedded Queries := Language Embedded Structure

```
Query(
  toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
)
```

AST / Intermediate Representation

```
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

Code Sample: EdslSimple.scala


```
Query(
  toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
)
```

```
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

```
trait Ast
object Ast:
  case class Subtract(a: Ast, b: Ast) extends Ast
  case class Concat(a: Ast, b: Ast) extends Ast
  case class ToString(v: Ast) extends Ast
  case class Constant(v: String) extends Ast
  case class Key(name: String) extends Ast
```

Code Sample: EdslSimple.scala


```
Query(
  toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
)
```

```
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

```
SELECT toString(data.underlyingPrice - data.strikePrice) +
  '/' + data.symbol
```

Code Sample: EdslSimple.scala

From Here

To Here

```
Query(
  toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
)
```

Serialize to this

```
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

```
{
  "Concat" : { "left" : {
      "Concat" : { "left" : {
          "ToString" : {"value" : {
              "Subtract" : {
                "left" : { "Key" : { "name" : "underlyingPrice" } },
                "right" : { "Key" : { "name" : "strikePrice" } }
              }
            }
          }
        }, "right" : { "Constant" : { "value" : "/" } }
      }
    }, "right" : { "Key" : { "name" : "symbol" } }
  }
}
```

Code Sample: EdslSimple.scala

Serialize to this

```
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

Deserialize back

```
{
  "Concat" : { "left" : {
      "Concat" : { "left" : {
          "ToString" : {"value" : {
              "Subtract" : {
                "left" : { "Key" : { "name" : "underlyingPrice" } },
                "right" : { "Key" : { "name" : "strikePrice" } }
              }
            }
          }
        }, "right" : { "Constant" : { "value" : "/" } }
      }
    }, "right" : { "Key" : { "name" : "symbol" } }
  }
}
```

```
// Deserialized on server
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

evaluate using

```
// Deserialized on server
Query(
  Concat(
    Concat(
      ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
      Constant("/")
    ),
    Key("symbol")
  )
)
```

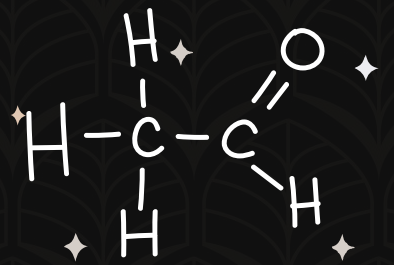
```
case class Evaluator(row: Map[String, Any]):
  private def eval(ast: Ast): Any =
    ast match
      case Subtract(a: Ast, b: Ast) => eval(a).toInt - eval(b).toInt
      case Concat(a: Ast, b: Ast)   => eval(a).toString + eval(b).toString
      case ToString(v: Ast)         => eval(v).toString
      case Constant(v: String)      => v
      case Key(value: String)       => row(value)

  def apply(q: Query) =
    eval(q.ast)
```

result

```
21/AAPL
117/MSFT
54/NFLX
136/GOOG
```

Code Sample: EdslSimple.scala

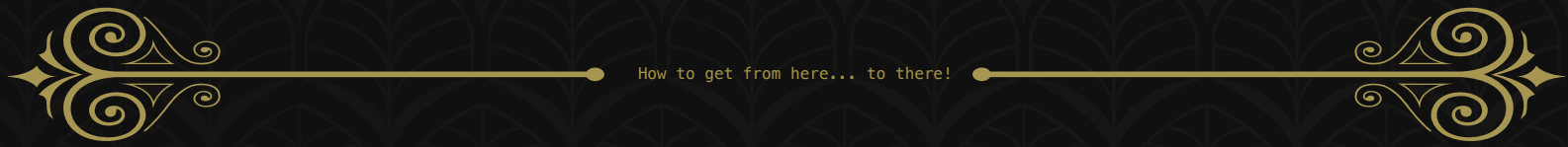


02

Embedded DSLs (EDSLs)

The Good, The Bad, and The Generic

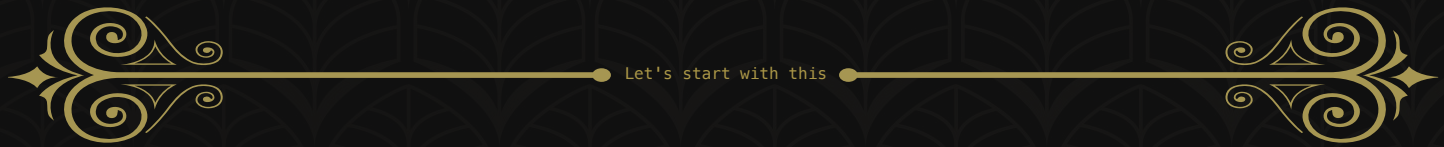
```
toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
```




```
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

```
toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
```

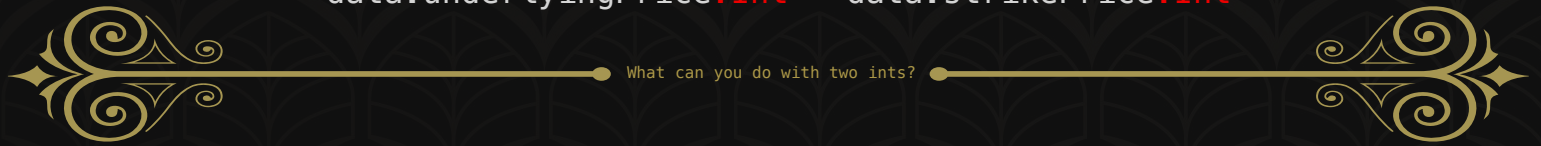



```
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

```scala
case class RemoteData(
  symbol: String
  strikePrice: Int
  underlyingPrice: Int
)
```

```
data.underlyingPrice:Int - data.strikePrice:Int
```



Code Sample: UseEdsl.scala

```scala
case class RemoteData(
  symbol: String
  strikePrice: Int
  underlyingPrice: Int
)
```

Int

Not much! You can't see inside!

Code Sample: UseEdsl.scala

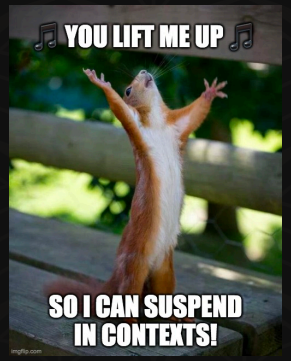
```scala
case class RemoteData(
  symbol: Value[String],
  strikePrice: Value[Int],
  underlyingPrice: Value[Int]
) {
  def map: Map[String, Any] = _
}

case class Value[T](key: String, value: T)
```

Typing Makes EDSLs Easier

```scala
trait Ast[T]
object Ast:
  case class Subtract(a: Ast[Int], b: Ast[Int]) extends Ast[Int]
  case class Concat(a: Ast[String], b: Ast[String]) extends Ast[String]
  case class ToString(v: Ast[Int]) extends Ast[String]
  case class Constant(v: String) extends Ast[String]
  case class Key[T](name: String) extends Ast[String]
```

Let's Pre-Lift Everything...



Code Sample: UseEdsl.scala

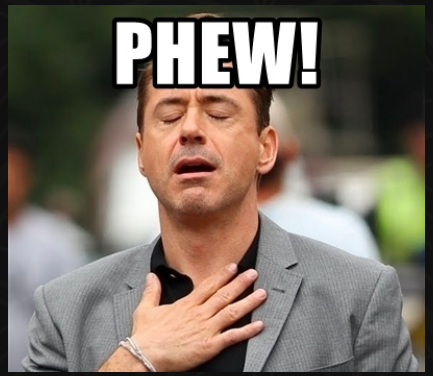
```scala
RemoteData(
  Value("symbol", "AAPL"),
  Value("strikePrice", 123),
  Value("underlyingPrice", 456)
) {
  def map: Map[String, Any] =
    Map("symbol" -> "AAPL", "strikePrice" -> 123, "underlyingPrice" -> 456)
}
```

Let's Pre-Lift Everything...

Code Sample: UseEdsl.scala

```
RemoteData.Construct(
  Value("AAPL"),
  Value(123),
  Value(456)
) {
  def map: Map[String, Any] = makeMap()
}
```

Let's Pre-Lift Everything...



Code Sample: UseEdsl.scala

```
RemoteData.Construct(
  Value("AAPL"),
  Value(123),
  Value(456)
) {
  def map: Map[String, Any] = makeMap()
}
```

```
toString(data.underlyingPrice - data.strikePrice) +
  "/" +
  data.symbol
```


```
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

```
RemoteData.Construct(
  Value("AAPL"),
  Value(123),
  Value(456)
) {
  def map: Map[String, Any] = makeMap()
}
```

```
toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]) +
  "/" +
  data.symbol
```

Not your grand-daddy's integers!

```
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

Capture! Not Evaluate!

```scala
extension (v: Value[Int])
  def -(other: Value[Int]) = Subtract(Key[Int](v.key), Key[Int](other.key))
```

```scala
toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]) +
  "/" +
  data.symbol
```

♫ I'm extending away ♫ ...

```scala
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

Define a DSL Expression

```scala
extension (v: Value[Int])
  def -(other: Value[Int]): Ast[Int] = Subtract(Key[Int](v.key), Key[Int](other.key))
```

```scala
toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]) +
  "/" +
  data.symbol
```

...'Cause I've got to make AST 🎵

```scala
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

Just a method...

```scala
def toString(ast: Ast[Int]):Ast[String] = ToString(ast)
```

```scala
toString((data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]) :Ast[Int]) +
  "/" +
  data.symbol
```

...and to track the method-call history ♫

```scala
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

Define a DSL Expression

```scala
extension (v: Ast[String])
  def +(other: String): Ast[String] = Concat(v, Constant(other))
```

```scala
toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]):Ast[String] +
  "/" +
  data.symbol
```

...and to track the method-call history ♫

```scala
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

Define a DSL Expression

```scala
extension (v: Ast[String])
  def +(other: Value[String]): Ast[String] = Concat(v, Key(other.key))
```

```scala
(toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]) +
  "/"):Ast[String] +
  data.symbol:Value[String]
```

...and to track the method-call history ♫

```scala
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

```scala
extension (v: Value[Int])
  def -(other: Value[Int]) = Subtract(Key(v.key), Key(other.key))

extension (v: Value[String])
  def +(other: Value[String]) = Concat(Key(v.key), Key(other.key))

extension (v: Ast[String])
  def +(other: Value[String]): Ast[String] = Concat(v, Key(other.key))
  def +(other: String): Ast[String] = Concat(v, Constant(other))

def toString(ast: Ast[Int]) = ToString(ast)
```

```scala
(toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]) +
  "/"):Ast[String] +
  data.symbol:Value[String]
```

... and it is done! Done! Huzzah!

```scala
Concat(
  Concat(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    Constant("/")
  ),
  Key("symbol")
)
```

Code Sample: UseEdsl.scala

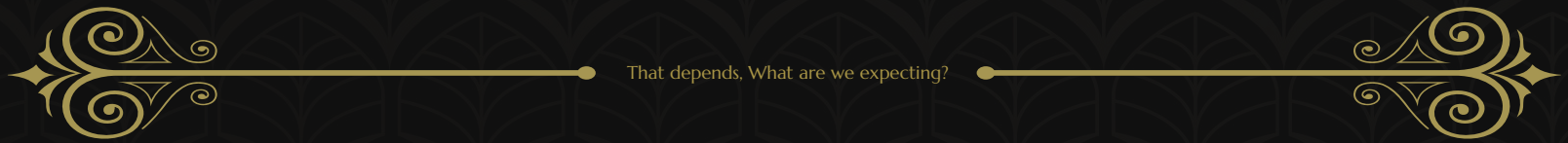
Types are a bit complex?

```
(toString(data.underlyingPrice:Value[Int] - data.strikePrice:Value[Int]):Ast[String] +
  "/"):Ast[String] +
  data.symbol:Value[String]
```

Code Sample: UseEdsl.scala

Can we implement `map`?

```
toString(data.underlyingPrice - data.strikePrice) // Ast[String]
  .map((value: String) => value + "/")
```



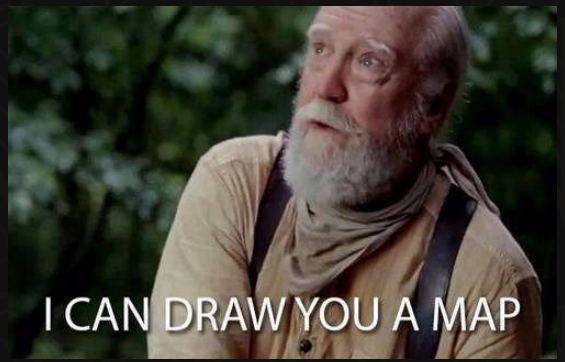

```
Map(
  ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
  ???
)
```

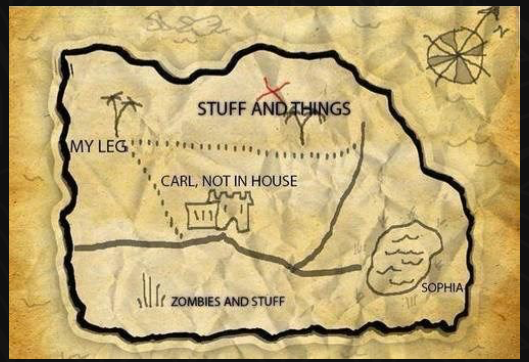
Code Sample: UseEdsl.scala

Can we implement `map`?

```
toString(data.underlyingPrice - data.strikePrice): Ast[String]
  .map((value: String) => value + "/")
```

If we just have really low expectations...

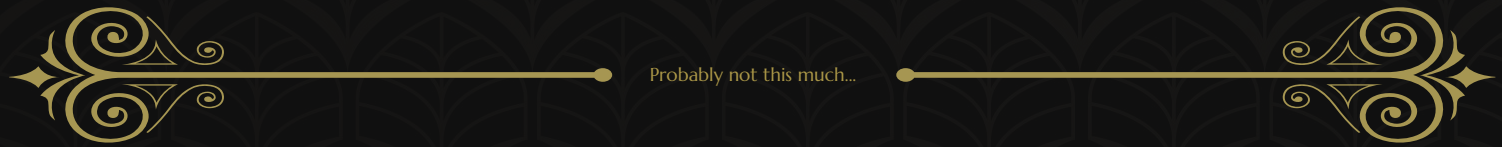




Code Sample: UseEdsl.scala

Can we implement `map`? We can't see inside!

```
toString(data.underlyingPrice - data.strikePrice): Ast[String]
  .map((value: String) => value + "/"): Ast[String]
```

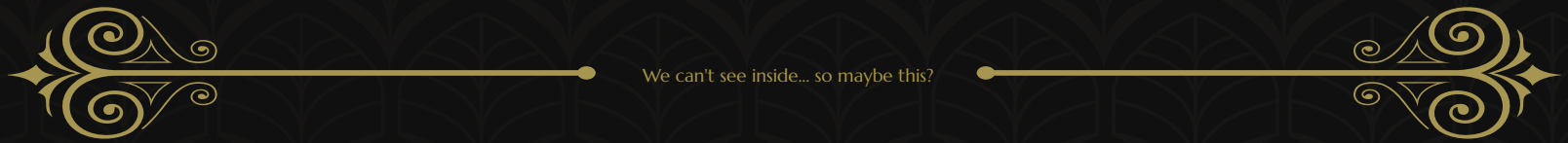



```
Map(
  ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
  SetVariable("value"),
  Concat(GetVariable("value"), Constant("/"))
)
```

Code Sample: UseEdsl.scala

Can we implement `map`? We can't see inside!

```
toString(data.underlyingPrice - data.strikePrice): Ast[String]
  .map(String => String): Ast[String]
```

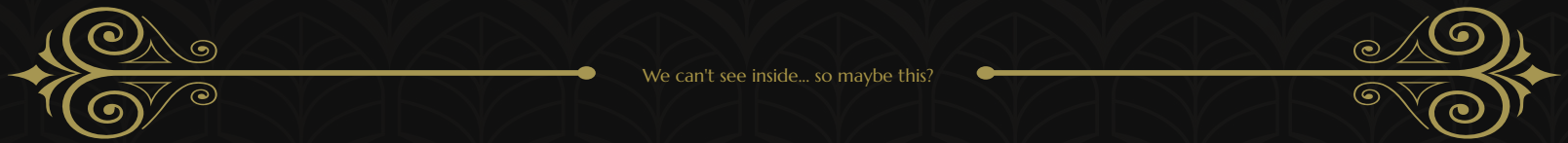



```
Map(
  ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
  f: String => String
)
```

Code Sample: UseEdsl.scala

Can we implement `map`? We can't see inside!

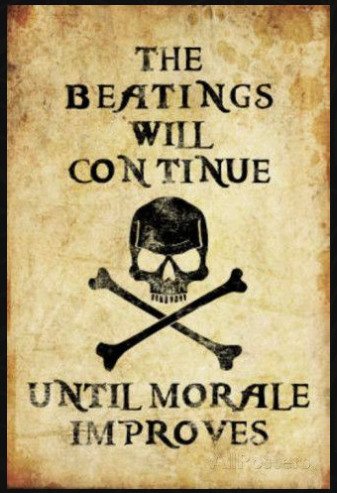
```
toString(data.underlyingPrice - data.strikePrice): Ast[String]
  .map(String => String): Ast[String]
```




```
trait Ast[T]
object Ast:
  case class Subtract(a: Ast[Int], b: Ast[Int]) extends Ast[Int]
  case class Concat(a: Ast[String], b: Ast[String]) extends Ast[String]
  case class ToString(v: Ast[Int]) extends Ast[String]
  case class Constant(v: String) extends Ast[String]
  case class Key[T](name: String) extends Ast[String]
  case class Map[T, R](ast: Ast[T])(f: T => R) extends Ast[T]
```

Code Sample: UseEdsl.scala

That's All Folks?




```
// Other possible encodings?
extension [T](value: Ast[T])
  def map[R](f: Key[T] => Ast[R]): Ast.Container[R]

// Using Typeclasses to mediate?
extension [T](value: Ast[T])
  def map[R](f: T => R)(implicit c: Container[T, R]): Ast[R]
```

03

Quoted-DSLs (QDSLs)

The Long but Short Path

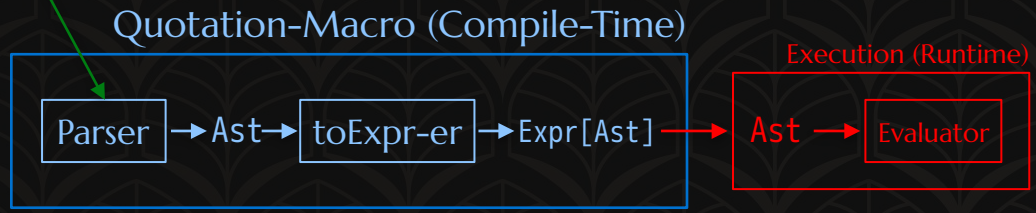
Quoted-DSL := A DSL Passed into a Quotation-Macro

Expression

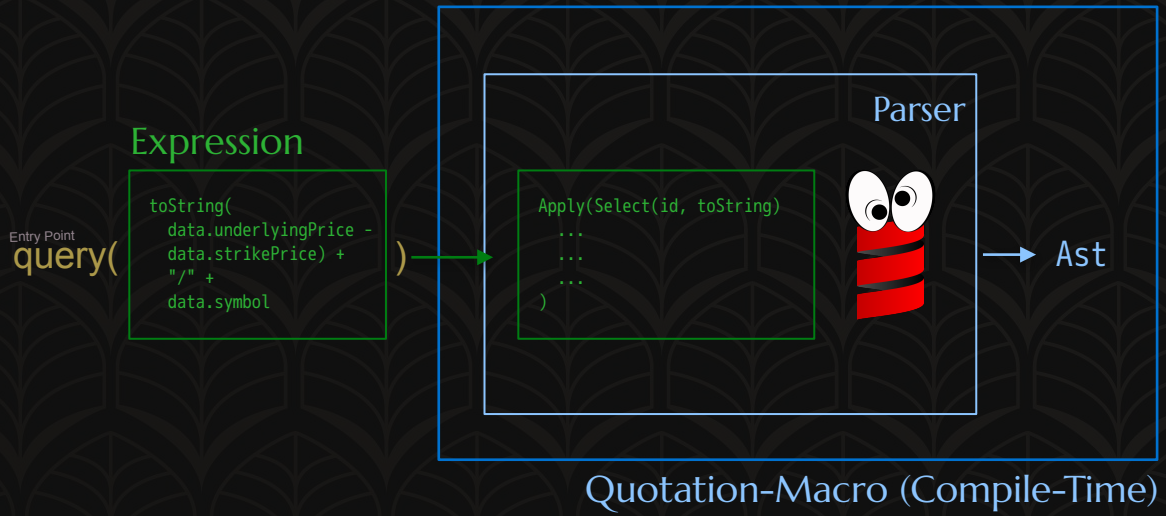
```
toString(
  data.underlyingPrice -
  data.strikePrice) +
  "/" +
  data.symbol
```

Ast i.e. OUR syntax tree

```
trait Ast
object Ast:
  case class Subtract(a: Ast, b: Ast) extends Ast
  case class Concat(a: Ast, b: Ast) extends Ast
  case class ToString(v: Ast) extends Ast
  case class Constant(v: String) extends Ast
  case class Key(name: String) extends Ast
```

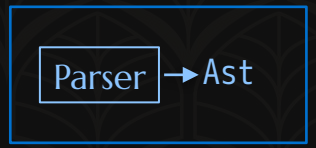


Quoted-DSL := A DSL Passed into a Quotation-Macro



The World through a Compiler's Eyes

```
val data: RemoteData = ...
```




```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
data.underlyingPrice(foo)
Apply(Select(Ident("data"), "underlyingPrice"), List(foo))
```

```
data.underlyingPrice("foo")
Apply(Select(Ident("data"), "underlyingPrice"), List(Constant("foo")))
// actually Literal(Constant("foo"))
```

Match the Expression

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  value.asTerm match
    case Select(Ident("data"), "underlyingPrice") =>
      Ast.Key("underlyingPrice")

inline def query(value: Any): Ast = ${ queryImpl('value) }
// In Quill this is called `quote`
```


```
> query(data.underlyingPrice: String Value[String])
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Generalize to Any Key

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  value.asTerm match
    case Select(Ident(data), underlyingPrice) =>
      Ast.Key(underlyingPrice)

inline def query(value: Any): Ast = ${ queryImpl('value) }
```

```
> query(data.underlyingPrice: String ~~Value[String]~~)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Make sure the Type is Right

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  value.asTerm match
    case Select(Ident(data), underlyingPrice) if (is[RemoteData](id.tpe)) =>
      Key(underlyingPrice)

inline def query(value: Any): Ast = ${ queryImpl('value) }
```

```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Make `.asTerm` into an Extractor

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  value match
    case AsTerm(Select(Ident(data), underlyingPrice)) if (is[RemoteData](id.tpe)) =>
      Key(underlyingPrice)

inline def query(value: Any): Ast = ${ queryImpl('value) }
```

```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

...

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  value match
    case AsTerm(Select(Ident(data), underlyingPrice)) if (is[RemoteData](id.tpe)) =>
      Key(underlyingPrice)

inline def query(value: Any): Ast = ${ queryImpl('value) }
```

```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

... let's make some space

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  value match
    case AsTerm(Select(Ident(data), underlyingPrice)) if (is[RemoteData](id.tpe)) =>
      Key(underlyingPrice)

inline def query(value: Any): Ast = ${ queryImpl('value) }
```

```
> query(data.underlyingPrice)
Ast.Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Introduce a Recursive method to Parse

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
    }
  val ast: Ast = recurse(value)
```

```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

... and a catch-all case that tells us what we missed

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
```

```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Introduce a recursive method to parse

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
```


```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Introduce a recursive method to parse

```
data.underlyingPrice
Select(Ident("data"), "underlyingPrice")
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(data.underlyingPrice)
Key("underlyingPrice")
```

Code Sample: Qdsl.scala

Let's do that recursive parsing now!

```
data.underlyingPrice.-(data.strikePrice)
Apply(Select(left, "-"), List(right))
// '{ ($left: Int) - ($right: Int) }
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(Select(left, "-"), List(right))) =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(data.underlyingPrice - data.strikePrice)
Subtract(Key("underlyingPrice") - Key("strikePrice"))
```

Code Sample: Qdsl.scala

Wait a second...

```
data.underlyingPrice - data.strikePrice
Apply(Select(left, "-"), List(right))
// '{ ($left: Int) - ($right: Int) }
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(Select(left, "-"), List(right))) =>
        Subtract(recurse(left:Term), recurse(right:Term))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(data.underlyingPrice - data.strikePrice)
Subtract(Key("underlyingPrice") - Key("strikePrice"))
```

Code Sample: Qdsl.scala

Terms and Exprs don't mix!

```
data.underlyingPrice - data.strikePrice
Apply(Select(left, "-"), List(right))
// '{ ($left: Int) - ($right: Int) }
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(Select(left, "-"), List(right))) =>
        Subtract(recurse(left.asExpr), recurse(right.asExpr))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(data.underlyingPrice - data.strikePrice)
Subtract(Key("underlyingPrice") - Key("strikePrice"))
```

Code Sample: Qdsl.scala

Let's Stay in Expr Land!

```
data.underlyingPrice - data.strikePrice
Apply(Select(left, "-"), List(right))
// '{ ($left: Int) - ($right: Int) }
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case '{ ($left: Int) - ($right: Int) } =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(data.underlyingPrice - data.strikePrice)
Subtract(Key("underlyingPrice") - Key("strikePrice"))
```

Code Sample: Qdsl.scala

Method calls work the same way

```
QDSL.toString(data.underlyingPrice - data.strikePrice)
Apply(Select(Ident("QDSL"), "toString"), List(content))
// '{ toString($content) }
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case '{ toString($i) } => ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(toString(data.underlyingPrice - data.strikePrice))
ToString(Subtract(Key("underlyingPrice") - Key("strikePrice")))
```

Code Sample: Qdsl.scala

Method calls work the same way

```
toString(data.underlyingPrice - data.strikePrice) + "/"
Apply(Select(left, "+"), List(right))
// '{ ($left: String) + ($right: String) }
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case '{ ($left: String) + ($right: String) } => Concat(recurse(left), recurse(right))
      case '{ toString($i) } => ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(toString(data.underlyingPrice - data.strikePrice) + "/")
Concat(ToString(Subtract(Key("underlyingPrice") - Key("strikePrice"))), Constant("/"))
```

Code Sample: Qdsl.scala

Let's have a look at constants...

```
"/"
Literal(StringConstant("/"))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Concat(recurse(left), recurse(right))
      case '{ toString($i) } => ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query("/")
Constant("/")
```

Code Sample: Qdsl.scala

In Expr-land, do it like this:

```
"/"
Literal(StringConstant("/")))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case '{ ${ Expr(value) }: String } => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Concat(recurse(left), recurse(right))
      case '{ toString($i) } => ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query("/")
Constant("/")
```

Code Sample: Qdsl.scala

We know Constants at compile-time

```
"/"
Literal(StringConstant("/"))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Literal(StringConstant(value:String))) => Ast.Constant(value:String)
      case '{ ($left: String) + ($right: String) } => Concat(recurse(left), recurse(right))
      case '{ toString($i) } => ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr.show}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(toString(data.underlyingPrice - data.strikePrice) + "/")
Concat(ToString(Subtract(Key("underlyingPrice") - Key("strikePrice"))), Constant("/"))
```

Code Sample: Qdsl.scala

Left-Hand is Scala's Syntax-Tree
Right-Hand is OUR Syntax-Tree (Ast)

```
"/"
Literal(StringConstant("/")))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(toString(data.underlyingPrice - data.strikePrice) + "/")
Ast.Concat(Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice") - Ast.Key("strikePrice"))), Ast.Constant("/"))
```

Code Sample: Qdsl.scala

All Simple Scala Types!

```
"/"
Literal(StringConstant("/")))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(toString(data.underlyingPrice:Int - data.strikePrice:Int):String + "/")
Ast.Concat(Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice") - Ast.Key("strikePrice"))), Ast.Constant("/"))
```

Code Sample: Qdsl.scala

Let's Try that Map Function Again

```
toString(data.underlyingPrice - data.strikePrice)
  .map(value => value + "/")
```

Code Sample: Qdsl.scala

Wait a minute....

```
toString(data.underlyingPrice - data.strikePrice): String
  .map((value: Char) => value + "/")
```

Behold! A ~~Phantom~~ Interpreted-Only Container!

```
Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/")
```


```
trait Entity[T]:
  def map[R](f: T => R): Entity[R]
object Entity:
  def apply[T](value: T): Entity[T] = ???
```

Code Sample: UseQdsl.scala

We don't *want* an implementation!

```
> Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/") ...BOOM!

query(Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/"))
```

Don't try this at home... err... outside a macro...

```
trait Entity[T]:
  def map[R](f: T => R): Entity[R]
object Entity:
  def apply[T](value: T): Entity[T] =
    throw new BOOM!("Should be interpreted by a macro. Not evaluated")
```

Code Sample: UseQdsl.scala

So we know this part...

```
Entity.apply(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/")
```


```
Apply(Select(Ident("Entity"), "apply"),
  List(
    Apply(Select(Select(Ident("data"), "underlyingPrice"), "-"), List(Select(Ident("data"), "strikePrice")))
  )
)
```

Code Sample: UseQdsl.scala

...and we sorta know this part...

```
Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/")
```


```
Apply(Select(Something("value"), "+"), Literal(Constant("/")))
```

Code Sample: UseQdsl.scala

...and we sorta know this part...

```
Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/")
```


```
Apply(Select(Something("value"), "+"), Literal(Constant("/")))
```

Code Sample: UseQdsl.scala

...what about this?

```
left
  .map((value: String) => value + "/")
```

... or trade it all in for what's in the mystery box!

```
Apply(Select(left, "map"),
  List(
    ???
  )
)
```

Code Sample: UseQdsl.scala

...what about this?

```
Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/")
```

... or trade it all in for what's in the mystery box!

```
Apply(Select(left, "map"),
  List(
    Lambda1(value, *Apply(Select(Ident(value), "+"), List(Constant("/"))))
  )
)
```

Code Sample: UseQdsl.scala

...what about this?

```
Entity(toString(data.underlyingPrice - data.strikePrice))
  .map((value: String) => value + "/")
```

... or trade it all in for what's in the mystery box!

```
Apply(Select(left, "map"),
  List(
    Lambda1(value, *Apply(Select(Ident(value), "+"), List(Constant("/"))))
  )
)
```

```
TypeApply(Apply(Select(stuff, "map"), List(Lambda1(Ident(value), stuffInside))))
```

Note: This slide was skipped in the presentation

What is actually happening here?

```scala
trait Ast
object Ast:
  ...
  case class Map(left: Ast, v: SetVariable, body: Ast) extends Ast
  case class SetVariable(name: String) extends Ast
  case class GetVariable(name: String) extends Ast
```

```scala
stuff
  .map(([SetVariable](value)) => [GetVariable](value) + "/")
```

Behold... yet another instruction set!

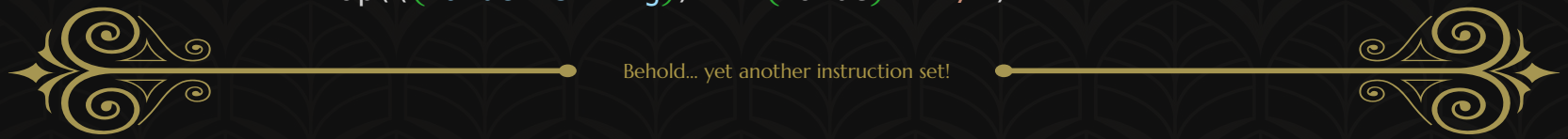
```scala
Apply(Select(left, "map"),
  List(
    Lambda1(value, Apply(Select(Ident(value), "+"), List(Constant("/"))))
  )
)
```

Code Sample: UseQdsl.scala

What is actually happening here?

```scala
trait Ast
object Ast:
  ...
  case class Map(left: Ast, v: SetVariable, body: Ast) extends Ast
  case class SetVariable(name: String) extends Ast
  case class GetVariable(name: String) extends Ast
```

```scala
stuff
  .map(((value: String)) => (value) + "/")
```




```scala
parse(Apply(Select(left, "map"),
  List(
    Lambda1(value, Apply(Select(Ident(value), "+"), List(Constant("/"))))
  )
)) => Ast.Map(
        left,
        Ast.SetVariable("value"),
        Ast.Concat(Ast.GetVariable("value"), Ast.Constant("/"))
      )
```

Code Sample: UseQdsl.scala

We know everything inside the Map Lambda!

```
stuff.map((value: String) => stuffInside)
Apply(Select(stuff, "map"), List(Lambda1(value, stuffInside)))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(TypeApply(Select(left, "map"), List(Lambda1(Ident(value), lambdaBody)), _))) =>
        Map(recurse(left.asExpr), SetKey(value), recurse(lambdaBody.asExpr))
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(Entity(toString(data.underlyingPrice - data.strikePrice))).map(value => value + "/")
Map(
  ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
  SetKey("value"), Concat(Key("value"), Constant("/"))
)
```

Code Sample: Qdsl.scala

...and that's a wrap?

```
stuff.map((value: String) => stuffInside)
Apply(Select(stuff, "map"), List(Lambda1(value, stuffInside)))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(TypeApply(Select(left, "map"), List(Lambda1(Ident(value), lambdaBody)), _))) =>
        Ast.Map(recurse(left.asExpr), SetKey(value), recurse(lambdaBody.asExpr))
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

```
> query(Entity(toString(data.underlyingPrice - data.strikePrice))).map(value => value + "/")
Ast.Map(
  Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice"), Ast.Key("strikePrice"))),
  Ast.SetKey("value"), Ast.Concat(Ast.Key("value"), Ast.Constant("/"))
)
```

Code Sample: Qdsl.scala

Wait a minute...

```
stuff.map((value: String) => stuffInside)
Apply(Select(stuff, "map"), List(Lambda1(value, stuffInside)))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(TypeApply(Select(left, "map"), List(Lambda1(Ident(value), lambdaBody)), _))) =>
        Ast.Map(recurse(left.asExpr), SetKey(value), recurse(lambdaBody.asExpr))
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($lef
        Ast.Subtrac
      case AsTerm(S                                                    =>
        Ast.Key(fi
      case _ =>
        report    owError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ ??? }
```

How do you do this???

```
> query(Entity(toString(data.underlyingPrice - data.strikePrice))).map(value => value + "/")
Ast.Map(
  Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice"), Ast.Key("strikePrice"))),
  Ast.SetKey("value"), Ast.Concat(Ast.Key("value"), Ast.Constant("/"))
)
```

Code Sample: Qdsl.scala

So... toExpr-ing?

```scala
def toExpr(ast: Ast): Expr[Ast] =
  ast match
    case Subtract(a: Ast, b: Ast)  => '{ Subtract(${ toExpr(a) }, ${ toExpr(b) }) }
    case Concat(a: Ast, b: Ast)    => '{ Concat(${ toExpr(a) }, ${ toExpr(b) }) }
    case ToString(v: Ast)          => '{ ToString(${ toExpr(v) }) }
    case Constant(v: String)       => '{ Constant(${ Expr(v) }) }
    case Key(name: String)         => '{ Key(${ Expr(name) }) }
    case SetVariable(name: String) => '{ SetVariable(${ Expr(name) }) }
    case GetVariable(name: String) => '{ GetVariable(${ Expr(name) }) }
    case Map(left: Ast, v: SetVariable, body: Ast) =>
      '{ Map(${ lift(toExpr) }, ${ toExpr(v).asExprOf[SetVariable] }, ${ toExpr(body) }) }
```

Code Sample: Qdsl.scala

If you want to get fancy!

```scala
object ToExprAst:
  def apply(ast: Ast) = Expr(ast)

  given ToExpr[Ast] with
    def apply(v: Ast) =
      v match
        case v: Subtract    => Expr(v)
        ...

  given ToExpr[Subtract] with
    def apply(v: Subtract) = '{ Subtract(${ Expr(v.a) }, ${ Expr(v.b) }) }
  given ToExpr[Concat] with
    def apply(v: Concat) = '{ Concat(${ Expr(v.a) }, ${ Expr(v.b) }) }
  given ToExpr[ToString] with
    def apply(v: ToString) = '{ ToString(${ Expr(v.v) }) }
  given ToExpr[Constant] with
    def apply(v: Constant) = '{ Constant(${ Expr(v.v) }) }
  given ToExpr[Key] with
    def apply(v: Key) = '{ Key(${ Expr(v.name) }) }
  given ToExpr[SetVariable] with
    def apply(v: SetVariable) = '{ SetVariable(${ Expr(v.name) }) }
  given ToExpr[GetVariable] with
    def apply(v: GetVariable) = '{ GetVariable(${ Expr(v.name) }) }
  given ToExpr[Map] with
    def apply(v: Map) = '{ Map(${ Expr(v.body) }, ${ Expr(v.v) }, ${ Expr(v.left) }) }

end ToExprAst
```

Code Sample: Qdsl.scala

Then snap it in!

```
stuff.map((value: String) => stuffInside)
Apply(Select(stuff, "map"), List(Lambda1(value, stuffInside)))
```

```
def queryImpl(value: Expr[Any]): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(TypeApply(Select(left, "map"), List(Lambda1(Ident(value), lambdaBody)), _))) =>
        Ast.Map(recurse(left.asExpr), SetKey(value), recurse(lambdaBody.asExpr))
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  toExpr(ast)
```

```
> query(Entity(toString(data.underlyingPrice - data.strikePrice))).map(value => value + "/")
Ast.Map(
  Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice"), Ast.Key("strikePrice"))),
  Ast.SetKey("value"), Ast.Concat(Ast.Key("value"), Ast.Constant("/"))
)
```

Code Sample: Qdsl.scala

Remember we wanted a Query type?

```
stuff.map((value: String) => stuffInside)
Apply(Select(stuff, "map"), List(Lambda1(value, stuffInside)))
```

```
def queryImpl(value: Expr[Any]): Expr[Query] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case AsTerm(Apply(TypeApply(Select(left, "map"), List(Lambda1(Ident(value), lambdaBody)), _))) =>
        Ast.Map(recurse(left.asExpr), SetKey(value), recurse(lambdaBody.asExpr))
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ Query(${lift(ast)}) }
```

```
> query(Entity(toString(data.underlyingPrice - data.strikePrice))).map(value => value + "/")
Ast.Map(
  Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice"), Ast.Key("strikePrice"))),
  Ast.SetKey("value"), Ast.Concat(Ast.Key("value"), Ast.Constant("/"))
)
```

Code Sample: Qdsl.scala

We can make it typed too!

```
stuff.map((value: String) => stuffInside)
Apply(Select(stuff, "map"), List(Lambda1(value, stuffInside)))
```

```
def queryImpl(value: Expr[Any]): Expr[Query[T]] =
  def recurse[T: Type](curr: Expr[T]): Ast =
    curr match {
      case AsTerm(Apply(TypeApply(Select(left, "map"), List(Lambda1(Ident(value), lambdaBody)), _))) =>
        Ast.Map(recurse(left.asExpr), SetKey(value), recurse(lambdaBody.asExpr))
      case AsTerm(Literal(StringConstant(value))) => Ast.Constant(value)
      case '{ ($left: String) + ($right: String) } => Ast.Concat(recurse(left), recurse(right))
      case '{ toString($i) } => Ast.ToString(recurse(i))
      case '{ ($left: Int) - ($right: Int) } =>
        Ast.Subtract(recurse(left), recurse(right))
      case AsTerm(Select(id @ Ident(data), fieldName)) if (is[RemoteData](id.tpe)) =>
        Ast.Key(fieldName)
      case _ =>
        report.throwError(s"Cannot match the expression: ${curr}")
    }
  val ast: Ast = recurse(value)
  '{ Query[T](${lift(ast)}) }
```

```
> query(Entity(toString(data.underlyingPrice - data.strikePrice))).map(value => value + "/")
Query[Entity[String]](Ast.Map(
  Ast.ToString(Ast.Subtract(Ast.Key("underlyingPrice"), Ast.Key("strikePrice"))),
  Ast.SetKey("value"), Ast.Concat(Ast.Key("value"), Ast.Constant("/"))
))
```

Recap!

```
query(
  Entity(toString(data.underlyingPrice - data.strikePrice))
    .map((value: String) => value + "/")
)
```

```
Query(
  Map(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    SetKey("value"), Concat(Key("value"), Constant("/"))
  )
)
```

Code Sample: UseQdsl.scala

So What???

So What???

Recap!

```
query(
  Entity(toString(data.underlyingPrice - data.strikePrice))
    .map((value: String) => value + "/")
)
```

```
Query(
  Map(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    SetKey("value"), Concat(Key("value"), Constant("/"))
  )
)
```

Code Sample: UseQdsl.scala

1) Types are Simple!

```
query(
  Entity(toString(data.underlyingPrice:Int - data.strikePrice:Int):String):Entity[String]
    .map((value: String) => value:String + "/"):Entity[String]
)
```

```
Query(
  Map(
    ToString(Subtract(Key("underlyingPrice"), Key("strikePrice"))),
    SetKey("value"), Concat(Key("value"), Constant("/"))
  )
)
```

Code Sample: UseQdsl.scala

2) Catch any kind of unsupported behavior

```
query(
  "foo".asInstanceOf[Int]
)
```

```
[error] -- Error: /stuff/qdsls-presentation/src/main/scala/org/dsl/UseQdsl.scala:9:11
[error] |      query(
[error] |      ^
[error] |Cannot match the expression: TypeApply(Select(Literal(StringConstant("foo")), "asInstanceOf"), List(TypeIdent("Int")))
[error] |        "foo".asInstanceOf[Int]
[error] |      )
```

```
Cannot match the expression: TypeApply(Select(Literal(StringConstant("foo")), "asInstanceOf"),
List(TypeIdent("Int"))) bloop

def query(value: Any): Ast

View Problem (⌘.)    No quick fixes available

query(
  "foo".asInstanceOf[Int]
)
```

Code Sample: UseQdsl.scala

2) ...seriously, I'm not kidding!

```
query(
  Entity("foo").map(v => System.exit(0))
)
```

```
[error] -- Error: /stuff/qdsls-presentation/src/main/scala/org/dsl/UseQdsl.scala:9:11
[error] |      query(
[error] |      ^
[error] |Cannot match the expression: Apply(Select(Ident("System"), "exit"), List(Literal(IntConstant(0))))
[error] |        Entity("foo").map(v => System.exit(0))
[error] |      )
```

```
inline def query(inline value: Any): Ast = ${ queryImpl('value) }

def query(value: Any): Ast

Cannot match the expression: Apply(Select(Ident("System"), "exit"), List(Literal(IntConstant(0)))) bloop

View Problem (⌘.)   No quick fixes available
```

```
query(
  Entity("foo").map(v => System.exit(0))
)
```

Code Sample: UseQdsl.scala

2) You can even tell the user how to correct their mistake!

```
query(
  "123".asInstanceOf[Int]
)
```

```
def queryImpl(value: Expr[Any])(using Quotes): Expr[Ast] =
  def recurse(curr: Expr[_]): Ast =
    curr match {
      case '{ ($str: String).asInstanceOf[Int] } =>
        report.throwError("Cannot class-cast using this DSL. Use .toInt instead.")
```

```
[error] -- Error: /stuff/qdsls-presentation/src/main/scala/org/dsl/UseQdsl.scala:9:11
[error] |      query(
[error] |      ^
[error] |      Cannot class-cast using this DSL. Use .toInt instead.
[error] |        "123".asInstanceOf[Int]
[error] |      )
```

Code Sample: UseQdsl.scala

3) View the AST at Compile-Time!

```
query(
  asString(data.underlyingPrice - data.strikePrice) + "/" + data.symbol
)
```

```
def queryImpl(value: Expr[Any])(using Quotes): Expr[Ast] =
  ...
  val result: Ast = recurse(underlyingArgument(value))
  report.info(Format(result.toString), value)
  lift(result)
```

```
[info] -- Info: /stuff/qdsls-presentation/src/main/scala/org/dsl/UseQdsl.scala:10:8
[info] 10 |        asString(data.underlyingPrice - data.strikePrice) + "/" + data.symbol
[info]    |        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[info]    |        Concat(
[info]    |          Concat(
[info]    |            ToString(Subtract(Key(underlyingPrice), Key(strikePrice))),
[info]    |            Constant(/)
[info]    |          ),
[info]    |          Key(symbol)
[info]    |        )
```

Code Sample: UseQdsl.scala

Advantages of QDSLs

1) Simple Types
2) Compile-Time Validation
3) Full Expression Control
4) User-Feedback at Compile Time

Thank You!!!

John DeGoes
Adam Fraser
Sandra Wolf
Damian Reeves
Ziverge
Capital One
ZIO Community