Quote or be quote”d”

Denys Shabalin

s”hello $name”

SIP-11 Extensible String Interpolation Syntax

q”hello $name”

SIP-11 Extensible ~~String Interpolation~~ Syntax Quotation

understanding desugaring

```
    q"hello $name"

 => StringContext("hello ", "").q(name)

 => Quasiquote(StringContext("hello ", "")).q(name)
```

understanding desugaring

```
implicit class Quasiquote(ctx: StringContext) {
  object q {
    def apply[T](args: T*): Tree = ...
  }
}
```

Joy

a functional, stack-oriented programming language

Joy basics

```
> 1 1 +
2

> 1 2 =
false
```

```
> true false or
true

> 2 2 2 * *
8
```

Joy basics

```
> [1 2 3 + +] i
6

> [1 2 3] first
1

> [1 2 3] rest
[2 3]
```

Joy basics

```
> 2 [0 >] [true] [false] ifte
true

> [1 2 3] [1 +] map
[2 3 4]

> [1 2 3 4] [2 rem = 0] filter
[2 4]
```

Joy syntax trees

```
sealed trait Joy
object Joy {
  final case class Int(value: scala.Int) extends Joy
  final case class Bool(value: Boolean) extends Joy
  final case class Name(value: String) extends Joy
  final case class Quoted(elems: List[Joy]) extends Joy
  final case class Program(elems: List[Joy]) extends Joy
}
```

Joy parsing

```
object parse extends RegexParsers {
  val lexical = new StdLexical
  lexical.delimiters ++= List("[", "]")
  lexical.reserved   ++= List("true", "false", "+", "*", ...)

  def int  = (...).r                          ^^  Joy.Int(_.toInt)
  def name = (...).r                          ^^  Joy.Name
  def tru  = "true"                           ^^^ Joy.Bool(true)
  def fls  = "false"                          ^^^ Joy.Bool(false)
  def quot = "[" ~> rep(joy) <~ "]"           ^^  Joy.Quoted
  def op   = ("+"  | "*"  | ... )             ^^  Joy.Name
  def prog = rep(joy)                         ^^  Joy.Program
  def joy  = tru | fls | op | name | int | quoted
}
```

Basic quotation

```
implicit class JoyQuote(ctx: StringContext) {
  def j(args: Joy*): Joy = Joy.parse(ctx.parts.head).get
}
```

```
scala> println(j"1 2 3 + *".toString)
Joy(Int(1), Int(2), Int(3), Name(+), Name(*))
```

Pretty printing

```
sealed trait Joy {
  final override def toString = this match {
    case Joy.Int(value)     => value.toString
    case Joy.Bool(value)    => value.toString
    case Joy.Name(value)    => value
    case Joy.Quoted(elems)  => elems.mkString("[", " ", "]")
    case Joy.Program(elems) => elems.mkString("", " ", "")
  }
}

scala> println(j"1 2 3 + *".toString)
1 2 3 + *
```

Macro quotation

```
implicit class JoyQuote(val ctx: StringContext) {
  def j(args: Joy*): Joy = macro JoyQuoteImpl.apply
}

class JoyQuoteImpl(val c: Context) {
  import c.universe._

 lazy val q"$_($_(..${parts: List[String]})).j(..$args)" =
    c.macroApplication

  implicit def lift[J <: Joy]: Liftable[J] = Liftable {
    case Joy.Int(value)    => q"_root_.joy.Joy.Int($value)"
    case Joy.Bool(value)   => q"_root_.joy.Joy.Bool($value)"
    case Joy.Name(value)   => q"_root_.joy.Joy.Name($value)"
    case Joy.Quoted(joys)  => q"_root_.joy.Joy.Quoted($joys)"
    case Joy.Program(joys) => q"_root_.joy.Joy.Program($joys)"
  }

  def apply(args: Tree*) = lift(Joy.parse(parts.head).get)
}
```

Macro quotation

```
scala> j"1 2 3 + *"

// expands into

scala> Joy.Program(List(Joy.Int(1), Joy.Int(2), Joy.Int(3),
                        Joy.Name("+"), Joy.Name("*")))
```

Macro quotation

```
scala> val two = j"2"
```

```
scala> j"$two $two +"
2 2 +
```

Unquoting

```
class JoyQuoteImpl(val c: Context) {
  ...

  object Hole {
    val pat = java.util.regex.Pattern.compile(...)
    def apply(i: Int) = s"$$placeholder$i"
    def unapply(s: String): Option[Int] = ...
  }

  def code() =
    parts.init.zipWithIndex.map { case (part, i) =>
      s"$part${Hole(i)}"
    }.mkString("", "", parts.last)

  implicit def lift[J <: Joy]: Liftable[J] = ...

  def apply(args: Tree*) = lift(Joy.parse(code()).get)
}
```

Unquoting

```
scala> val two = j"2"
two: joy.Joy.Int = 1

scala> j"$two $two +"
res0: joy.Joy.Program = $placeholder0 $placeholder1 +
```

Unquoting

```
implicit def lift[J <: Joy]: Liftable[J] = Liftable {
  case Joy.Int(value)    => q"_root_.joy.Joy.Int($value)"
  case Joy.Bool(value)   => q"_root_.joy.Joy.Bool($value)"
  case Joy.Name(Hole(i)) => args(i)
  case Joy.Name(value)   => q"_root_.joy.Joy.Name($value)"
  case Joy.Quoted(joys)  => q"_root_.joy.Joy.Quoted($joys)"
  case Joy.Program(joys) => q"_root_.joy.Joy.Program($joys)"
}
```

Unquoting

```
scala> val two = j"2"
two: joy.Joy.Int = 2

scala> j"$two $two +"
res0: joy.Joy.Program = 2 2 +

// expands into

scala> Joy.Program(List(two, two, Joy.Name("+")))
```

Lifting

```
scala> val two = 2
```

```
scala> j"$two $two +"
2 2 +
```

Lifting

```
trait Joy
object Joy {
  ...
  // typeclass-based conversion to Joy tree

  trait Lift[T] { def apply(value: T): Joy }

  object Lift { ... }
}
```

Lifting

```
class JoyQuoteImpl(val c: Context) {
  ...

  def arg(i: Int) = {
    val arg = args(i)
    val tpe = arg.tpe
    if (tpe <:< typeOf[Joy]) arg
    else {
      val LiftT = appliedType(typeOf[Joy.Lift[_]], tpe)
      val lift = c.inferImplicitValue(LiftT, silent = true)
      if (lift.nonEmpty) q"$lift($arg)"
      else c.abort(...)
    }
  }

  ...
}
```

Lifting

```
scala> val two = 2
ft: Int = 42
```

```
scala> j"$two $two +"
res1: joy.Joy.Program = 2 2 +
```

Splicing

```
scala> val xs = List(j"a", j"b", j"c")

scala> j"..$xs + +"
a b c + +
```

Splicing

```
implicit class JoyQuote(val ctx: StringContext) {
  def j[T](args: T*): Joy = macro JoyQuoteImpl.apply
}

class JoyQuoteImpl(val c: Context) {
  import c.universe._

  lazy val q"$_($_(..${parts: List[String]})).j[..$_](..$args)" =
c.macroApplication
  ...
}
```

Splicing

```
class JoyQuoteImpl(val c: Context) {
  ...

  implicit def liftJoys: Liftable[List[Joy]] = Liftable { joys =>
    def prepend(joys: List[Joy], t: Tree) = ...
    def append(t: Tree, joys: List[Joy]) = ...

    val (pre, middle) = joys.span(_ != Joy.Name(".."))
    middle match {
      case Nil =>
        prepend(pre, q"$Nil")
      case Joy.Name("..") :: Joy.Name(Hole(i)) :: rest =>
        append(prepend(pre, arg(i)), rest)
      case _ =>
        c.abort(...)
    }
  }

  implicit def lift[J <: Joy]: Liftable[J] = ...

  ...
}
```

Splicing

```
scala> val xs = List(j"a", j"b", j"c")
xs: List[joy.Joy.Name] = List(a, b, c)
```

```
scala> j"..$xs + +"
res1: joy.Joy.Program = a b c + +
```

Feature interaction: splicing of trees as collections

```
scala> val xs = j"1 2 3"
xs: joy.Joy.Program = 1 2 3

scala> j"..$xs + +"
<console>:12: error: .. value must be of iterable type
              j"..$xs + +"
              ^
```

Feature interaction: splicing of trees

```
object Joy {
  final case class Int(value: scala.Int) extends Joy
  final case class Bool(value: scala.Boolean) extends Joy
  final case class Name(value: String) extends Joy
  final case class Quoted(elems: List[Joy])
                extends Iterable[Joy] with Joy {
    def iterator = elems.iterator
  }
  final case class Program(elems: List[Joy])
                extends Iterable[Joy] with Joy {
    def iterator = elems.iterator
  }
}
```

Feature interaction: splicing of trees

```
scala> val xs = j"1 2 3"
xs: joy.Joy.Program = 1 2 3
```

```
scala> j"..$xs + +"
res0: joy.Joy.Program = 1 2 3 + +
```

Feature interaction: splicing + lifting

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> j"..$xs + +"
<console>:12: error: type mismatch;
 found   : joy.Joy
 required: List[joy.Joy]
              j"..$xs + +"
              ^
```

Feature interaction: splicing + lifting

```
class JoyQuoteImpl(val c: Context) {
  ...

  def arg(i: Int, dotted: Boolean = false) =  {
    val arg = args(i)
    val tpe = if (!dotted) arg.tpe else iterableT(arg.tpe)
    val subst: Tree => Tree =
      if (tpe <:< typeOf[Joy]) identity
      else {
        val LiftT = appliedType(typeOf[Joy.Lift[_]]
        val lift = c.inferImplicitValue(LiftT, tpe), silent = true)
        if (lift.nonEmpty) t => q"$lift($t)"
        else abort(s"couldn't find implicit value of type Lift[$tpe]"
      }
    if (!dotted) subst(arg)
    else {
      val x = TermName(c.freshName())
      q"$arg.map { ($x: $tpe) => ${subst(q"$x")} }.toList"
    }
  }


  ...
}
```

Feature interaction: splicing + lifting

```
class JoyQuoteImpl(val c: Context) {
  ...

  lazy val IterableClass: TypeSymbol =
    typeOf[Iterable[_]].typeSymbol.asType

  lazy val IterableTParam: Type =
    IterableClass.typeParams(0).asType.toType

  def iterableT(tpe: Type): Type =
    IterableTParam.asSeenFrom(tpe, IterableClass)

  ...
}
```

Feature interaction: splicing + lifting

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> j"..$xs + +"
res0: joy.Joy.Program = 1 2 3 + +
```

Status

so far we've obtained:

1. AST + parser
2. compile-time expression quotations
3. with higher-order unquoting
4. and typeclass-based interop through Lift[T]

looks good but we're not done yet

Pattern matching

```
scala> val j"1 2" = j"1 2"
<console>:10: error: macro method j is not a case class, nor does it
have an unapply/unapplySeq member
       val j"1 2" = j"1 2"
           ^
```

Pattern matching

```
implicit class JoyQuote(val ctx: StringContext) {
  object j {
    def apply[T](args: T*): Joy = macro JoyQuoteImpl.apply
    def unapply(scrutinee: Any): Any = macro JoyQuoteImpl.unapply
  }
}
class JoyQuoteImpl(val c: Context) {
  import c.universe._
  lazy val q"$_($_(..${parts: List[String]})).j.${method: TermName}[.
$_](..$args)" = c.macroApplication

  ...

  def expand = lift(Joy.parse(code()).get)
  def apply(args: Tree*) = expand
  def unapply(scrutinee: Tree) = expand
}
```

Pattern matching

```
scala> val j"1 2" = j"1 2"
// it works!
```

Pattern matching: unquoting

```
scala> val j"$a $b" = j"1 2"
java.lang.IndexOutOfBoundsException: 1
...
```

Pattern matching: unquoting

```
class JoyQuoteImpl(val c: Context) {
  ...
  def wrap(lifted: Tree): Tree = method match {
    case TermName("apply")   => lifted
    case TermName("unapply") =>
      val (thenp, elsep) =
        if (parts.length == 1) (q"true", q"false")
        else {
          val xs = ...
          (q"_root_.scala.Some((..$xs))", q"_root_.scala.None")
        }
      q""" new {
        def unapply(joy: _root_.joy.Joy) = {
          joy match {
            case $lifted => $thenp
            case _       => $elsep
          }
        }
      }.unapply(..$args) """
  }
  ...
}
```

Pattern matching: unquoting

```
class JoyQuoteImpl(val c: Context) {
  ...
  def arg(i: Int, dotted: Boolean = false) = method match {
    case TermName("apply") =>
      ...
    case TermName("unapply") =>
      val x = TermName(s"x$i")
      pq"$x @ _"
  }
  ...
}
```

Pattern matching: unquoting

```
scala> val j"$a $b" = j"1 2"
a: joy.Joy = 1
b: joy.Joy = 2
```

Pattern matching: splicing

```
scala> val j"$a ..$b" = j"1 2 3"
a: joy.Joy = 1
b: List[joy.Joy] = List(2, 3)
```

```
scala> val j"..$a $b" = j"1 2 3"
a: List[joy.Joy] = List(1, 2)
b: joy.Joy = 3
```

Pattern matching: (un)lifting

```
scala> val two = 2

scala> val sum = j"$two $two +"
```

```
scala> val j"${a: Int} ${b: Int} +" = sum
a: Int = 1
b: Int = 2
```

Pattern matching: (un)lifting

```
trait Joy
object Joy {
  ...

  // typeclass-based extraction out of Joy tree
  trait Unlift[T] { def apply(joy: Joy): Option[T] }

  object Unlift { ...}
}
```

Pattern matching: (un)lifting

```
class JoyQuoteImpl(val c: Context) {
  ...
    case TermName("unapply") =>
      val x = TermName(s"x$i")
      val subpattern = c.internal.subpatterns(args.head).get.apply(i)
      subpattern match {
        case pq"$_: $tpt" =>
          val tpe = c.typecheck(tpt, c.TYPEmode).tp
          val UnliftT = appliedType(typeOf[Joy.Unlift[_]], tpe)
          val unlift = c.inferImplicitValue(UnliftT, silent = true)
          if (unlift.isEmpty) pq"$unlift($x @ _)"
          else c.abort(...)
        case _ => pq"$x @ _"
      }
  ...
}
```

Pattern matching: (un)lifting

```
scala> val two = 2
two: Int = 2

scala> val sum = j"$two $two +"
sum: joy.Joy.Program = 2 2 +

scala> val j"${a: Int} ${b: Int} +" = sum
a: Int = 2
b: Int = 2
```

Pattern matching: fast forward to splicing + unlifting

```
with a bit more love one can get this working too:
```

```
scala> val j"..${xs: List[Int]} + +" = j"1 2 3 + +"
xs: List[Int] = List(1, 2, 3)
```

Status

```
j"" quotation:

1. works both as patterns and as expressions
2. supports both $regular unquoting and ..$splicing
3. supports customisable interop via Lift and Unlift

and all permutations (2^3) of these features
```

Eval

```
type Stack     = Joy.Quoted
type Transform = Stack => Stack

val lookup: Map[String, Transform] = Map(...)

implicit class Eval(joy: Joy) {
  def eval: Transform  = joy match {
    case Joy.Name(name)   =>
      lookup(name)

    case Joy.Program(ops) =>
      stack => ops.map(_.eval)
                  .reduce((a, b) => b compose a)
                  .apply(stack)

    case _                =>
      stack => j"[..$stack $joy]"
  }
}
```

Eval

```
val lookup: Map[String, Transform] = Map(
  ...

  "dup"  -> { case j"[..$init $x]"    => j"[..$init $x $x]"      },
  "swap" -> { case j"[..$init $a $b]" => j"[..$init $b $a]"      },
  "="    -> { case j"[..$init $a $b]" => j"[..$init ${a == b}]" },
  "!="   -> { case j"[..$init $a $b]" => j"[..$init ${a != b}]" },

  ...
)
```

Eval

```
val lookup: Map[String, Transform] = Map(
  ...

  "dup"  -> { case j"[..$init $x]"    => j"[..$init $x $x    ]" },
  "swap" -> { case j"[..$init $a $b]" => j"[..$init $b $a    ]" },
  "="    -> { case j"[..$init $a $b]" => j"[..$init ${a == b}]" },
  "!="   -> { case j"[..$init $a $b]" => j"[..$init ${a != b}]" },

  ...
  "not" -> { case j"[..$init ${b: Boolean}]" => j"[..$init ${!b}]" },
  ...
)

other default operations:
  and, or, >, >=, <, <=, +, -, *, rem, i,
  concat, size, first, rest, size, dip,
  map, filter, ifte, primrec, ...
```

Demo

Summary

1. Embedded languages via quotations are awesome
2. Feature interactions are hard,
   make sure you get it right from the first time
3. Macros aren't easy but this is changing
   (come to @xeno_by's talk to see it happen)

Q/A?