Numeric Literals

Note: This feature is not yet part of the Scala 3 language definition. It can be made available by a language import:

```
import scala.language.experimental.genericNumberLiterals
```

In Scala 2, numeric literals were confined to the primitive numeric types `Int`, `Long`, `Float`, and `Double`. Scala 3 allows to write numeric literals also for user-defined types. Example:

```
val x: Long = -10_000_000_000
val y: BigInt = 0x123_abc_789_def_345_678_901
val z: BigDecimal = 110_222_799_799.99

(y: BigInt) match
  case 123_456_789_012_345_678_901 =>
```

The syntax of numeric literals is the same as before, except there are no pre-set limits how large they can be.

Meaning of Numeric Literals

The meaning of a numeric literal is determined as follows:

- If the literal ends with `l` or `L`, it is a `Long` integer (and must fit in its legal range).
- If the literal ends with `f` or `F`, it is a single precision floating point number of type `Float`.
- If the literal ends with `d` or `D`, it is a double precision floating point number of type `Double`.

In each of these cases the conversion to a number is exactly as in Scala 2 or in Java. If a numeric literal does *not* end in one of these suffixes, its meaning is determined by the expected type:

1. If the expected type is `Int`, `Long`, `Float`, or `Double`, the literal is treated as a standard literal of that type.
2. If the expected type is a fully defined type `T` that has a given instance of type `scala.util.FromDigits[T]`, the literal is converted to a value of type `T` by passing it as an argument to the `fromDigits` method of that instance (more details below).
3. Otherwise, the literal is treated as a `Double` literal (if it has a decimal point or an exponent), or as an `Int` literal (if not). (This last possibility is again as in Scala 2 or Java.)

With these rules, the definition

```
val x: Long = -10_000_000_000
```

is legal by rule (1), since the expected type is `Long`. The definitions

```
val y: BigInt = 0x123_abc_789_def_345_678_901
val z: BigDecimal = 111222333444.55
```

are legal by rule (2), since both `BigInt` and `BigDecimal` have `FromDigits` instances (which implement the `FromDigits` subclasses `FromDigits.WithRadix` and `FromDigits.Decimal`, respectively). On the other hand,

```
val x = -10_000_000_000
```

gives a type error, since without an expected type `-10_000_000_000` is treated by rule (3) as an `Int` literal, but it is too large for that type.

The `FromDigits` Trait

To allow numeric literals, a type simply has to define a `given` instance of the `scala.util.FromDigits` type class, or one of its subclasses. `FromDigits` is defined as follows:

```
trait FromDigits[T]:
  def fromDigits(digits: String): T
```

Implementations of the `fromDigits` convert strings of digits to the values of the implementation type `T`. The `digits` string consists of digits between `0` and `9`, possibly preceded by a sign ("+" or "-"). Number separator characters `_` are filtered out before the string is passed to `fromDigits`.

The companion object `FromDigits` also defines subclasses of `FromDigits` for whole numbers with a given radix, for numbers with a decimal point, and for numbers that can have both a decimal point and an exponent:

```
object FromDigits:

  /** A subclass of `FromDigits` that also allows to convert whole    *  numbe
   */
  trait WithRadix[T] extends FromDigits[T]:
    def fromDigits(digits: String): T = fromDigits(digits, 10)
    def fromDigits(digits: String, radix: Int): T

  /** A subclass of `FromDigits` that also allows to convert number    *  liter
   */
  trait Decimal[T] extends FromDigits[T]

  /** A subclass of `FromDigits`that allows also to convert number    *  litera
   *  exponent `('e' | 'E')['+' | '-']digit digit*`.
   */
  trait Floating[T] extends Decimal[T]
```

A user-defined number type can implement one of those, which signals to the compiler that hexadecimal numbers, decimal points, or exponents are also accepted in literals for this type.

Error Handling

`FromDigits` implementations can signal errors by throwing exceptions of some subtype of `FromDigitsException`. `FromDigitsException` is defined with three subclasses in the `FromDigits` object as follows:

```
abstract class FromDigitsException(msg: String) extends NumberFormatException(r

class NumberTooLarge (msg: String = "number too large")         extends FromDig
class NumberTooSmall (msg: String = "number too small")         extends FromDig
class MalformedNumber(msg: String = "malformed number literal") extends FromDig
```

Example

As a fully worked out example, here is an implementation of a new numeric class, `BigFloat`, that accepts numeric literals. `BigFloat` is defined in terms of a `BigInt` mantissa and an `Int` exponent:

```
case class BigFloat(mantissa: BigInt, exponent: Int):
  override def toString = s"${mantissa}e${exponent}"
```

`BigFloat` literals can have a decimal point as well as an exponent. E.g. the following expression should produce the `BigFloat` number `BigFloat(-123, 997)`:

```
-0.123E+1000: BigFloat
```

The companion object of `BigFloat` defines an `apply` constructor method to construct a `BigFloat` from a `digits` string. Here is a possible implementation:

```
object BigFloat:
  import scala.util.FromDigits

  def apply(digits: String): BigFloat =
    val (mantissaDigits, givenExponent) =
      digits.toUpperCase.split('E') match
        case Array(mantissaDigits, edigits) =>
          val expo =
            try FromDigits.intFromDigits(edigits)
            catch case ex: FromDigits.NumberTooLarge =>
              throw FromDigits.NumberTooLarge(s"exponent too large: $edigits")
          (mantissaDigits, expo)
        case Array(mantissaDigits) =>
          (mantissaDigits, 0)
    val (intPart, exponent) =
      mantissaDigits.split('.') match
        case Array(intPart, decimalPart) =>
          (intPart ++ decimalPart, givenExponent - decimalPart.length)
        case Array(intPart) =>
          (intPart, givenExponent)
    BigFloat(BigInt(intPart), exponent)
```

To accept `BigFloat` literals, all that's needed in addition is a `given` instance of type `FromDigits.Floating[BigFloat]`:

```
given FromDigits: FromDigits.Floating[BigFloat] with
    def fromDigits(digits: String) = apply(digits)
end BigFloat
```

Note that the `apply` method does not check the format of the `digits` argument. It is assumed that only valid arguments are passed. For calls coming from the compiler that assumption is valid, since the compiler will first check whether a numeric literal has the correct format before it gets passed on to a conversion method.

Compile-Time Errors

With the setup of the previous section, a literal like

```
1e10_0000_000_000: BigFloat
```

would be expanded by the compiler to

```
BigFloat.FromDigits.fromDigits("1e100000000000")
```

Evaluating this expression throws a `NumberTooLarge` exception at run time. We would like it to produce a compile-time error instead. We can achieve this by tweaking the `BigFloat` class with a small dose of metaprogramming. The idea is to turn the `fromDigits` method into a macro, i.e. make it an inline method with a splice as right-hand side. To do this, replace the `FromDigits` instance in the `BigFloat` object by the following two definitions:

```
object BigFloat:
  ...

  class FromDigits extends FromDigits.Floating[BigFloat]:
    def fromDigits(digits: String) = apply(digits)

  given FromDigits with
    override inline def fromDigits(digits: String) = ${
      fromDigitsImpl('digits)
    }
```

Note that an inline method cannot directly fill in for an abstract method, since it produces no code that can be executed at runtime. That is why we define an intermediary class `FromDigits` that contains a fallback implementation which is then overridden by the inline method in the `FromDigits` given instance. That method is defined in terms of a macro implementation method `fromDigitsImpl`. Here is its definition:

```
private def fromDigitsImpl(digits: Expr[String])(using ctx: Quotes): Expr[BigF
    digits.value match
      case Some(ds) =>
```

```
        try
          val BigFloat(m, e) = apply(ds)
          '{BigFloat(${Expr(m)}, ${Expr(e)})}
        catch case ex: FromDigits.FromDigitsException =>
          ctx.error(ex.getMessage)
          '{BigFloat(0, 0)}
      case None =>
        '{apply($digits)}
end BigFloat
```

The macro implementation takes an argument of type `Expr[String]` and yields a result of type `Expr[BigFloat]`. It tests whether its argument is a constant string. If that is the case, it converts the string using the `apply` method and lifts the resulting `BigFloat` back to `Expr` level. For non-constant strings `fromDigitsImpl(digits)` is simply `apply(digits)`, i.e. everything is evaluated at runtime in this case.

The interesting part is the `catch` part of the case where `digits` is constant. If the `apply` method throws a `FromDigitsException`, the exception's message is issued as a compile time error in the `ctx.error(ex.getMessage)` call.

With this new implementation, a definition like

```
val x: BigFloat = 1234.45e3333333333
```

would give a compile time error message:

```
3 |  val x: BigFloat = 1234.45e3333333333
  |                    ^^^^^^^^^^^^^^^^^^
  |                    exponent too large: 3333333333
```

‹ Named... Explicit... ›