Rules for Operators

The rules for infix operators have changed in some parts:

First, an alphanumeric method can be used as an infix operator only if its definition carries an `infix` modifier.

Second, it is recommended (but not enforced) to augment definitions of symbolic operators with `@targetName` annotations.

Finally, a syntax change allows infix operators to be written on the left in a multi-line expression.

The `infix` Modifier

An `infix` modifier on a method definition allows using the method as an infix operation. Example:

```
import scala.annotation.targetName

trait MultiSet[T]:

  infix def union(other: MultiSet[T]): MultiSet[T]

  def difference(other: MultiSet[T]): MultiSet[T]

  @targetName("intersection")
  def *(other: MultiSet[T]): MultiSet[T]

end MultiSet

val s1, s2: MultiSet[Int]

s1 union s2         // OK
s1 `union` s2       // also OK but unusual
s1.union(s2)        // also OK
```

```
s1.difference(s2)   // OK
s1 `difference` s2  // OK
s1 difference s2    // gives a deprecation warning

s1 * s2             // OK
s1 `*` s2           // also OK, but unusual
s1.*(s2)            // also OK, but unusual
```

Infix operations involving alphanumeric operators are deprecated, unless one of the following conditions holds:

- the operator definition carries an `infix` modifier, or
- the operator was compiled with Scala 2, or
- the operator is followed by an opening brace.

An alphanumeric operator is an operator consisting entirely of letters, digits, the `$` and `_` characters, or any Unicode character `c` for which `java.lang.Character.isIdentifierPart(c)` returns `true`.

Infix operations involving symbolic operators are always allowed, so `infix` is redundant for methods with symbolic names.

The `infix` modifier can also be given to a type:

```
infix type or[X, Y]
val x: String or Int = ...
```

Motivation

The purpose of the `infix` modifier is to achieve consistency across a code base in how a method or type is applied. The idea is that the author of a method decides whether that method should be applied as an infix operator or in a regular application. Use sites then implement that decision consistently.

Details

1. `infix` is a soft modifier. It is treated as a normal identifier except when in modifier position.

2. If a method overrides another, their infix annotations must agree. Either both are annotated with `infix`, or none of them are.

3. `infix` modifiers can be given to method definitions. The first non-receiver parameter list of an `infix` method must define exactly one parameter

parameter list of an `infix` method must define exactly one parameter.
Examples:

```
infix def op1(x: S): R              // ok
infix def op2[T](x: T)(y: S): R     // ok
infix def op3[T](x: T, y: S): R     // error: two parameters

extension (x: A)
  infix def op4(y: B): R           // ok
  infix def op5(y1: B, y2: B): R   // error: two parameters
```

4. `infix` modifiers can also be given to type, trait or class definitions that have exactly two type parameters. An infix type like

   ```
   infix type op[X, Y]
   ```

   can be applied using infix syntax, i.e. `A op B`.

5. To smooth migration to Scala 3.0, alphanumeric operators will only be deprecated from Scala 3.1 onwards, or if the `-source future` option is given in Dotty/Scala 3.

The `@targetName` Annotation

It is recommended that definitions of symbolic operators carry a `@targetName` annotation that provides an encoding of the operator with an alphanumeric name. This has several benefits:

- It helps interoperability between Scala and other languages. One can call a Scala-defined symbolic operator from another language using its target name, which avoids having to remember the low-level encoding of the symbolic name.
- It helps legibility of stacktraces and other runtime diagnostics, where the user-defined alphanumeric name will be shown instead of the low-level encoding.
- It serves as a documentation tool by providing an alternative regular name as an alias of a symbolic operator. This makes the definition also easier to find in a search.

Syntax Change

Infix operators can now appear at the start of lines in a multi-line expression. Examples:

```
val str = "hello"
```

```
  ++ " world"
  ++ "!"


def condition =
  x > 0
  ||
  xs.exists(_ > 0)
  || xs.isEmpty
```

Previously, those expressions would have been rejected, since the compiler's semicolon inference would have treated the continuations `++ " world"` or `|| xs.isEmpty` as separate statements.

To make this syntax work, the rules are modified to not infer semicolons in front of leading infix operators. A *leading infix operator* is

- a symbolic identifier such as `+`, or `approx_==`, or an identifier in backticks that
- starts a new line, and
- is not following a blank line, and
- is followed by at least one whitespace character and a token that can start an expression.
- Furthermore, if the operator appears on its own line, the next line must have at least the same indentation width as the operator.

Example:

```
freezing
  | boiling
```

This is recognized as a single infix operation. Compare with:

```
freezing
  !boiling
```

This is seen as two statements, `freezing` and `!boiling`. The difference is that only the operator in the first example is followed by a space.

Another example:

```
println("hello")
  ???
  ??? match { case 0 => 1 }
```

This code is recognized as three different statements. `???` is syntactically a symbolic

This code is recognized as three different statements: `!!!` is syntactically a symbolic identifier, but neither of its occurrences is followed by a space and a token that can start an expression.

Unary operators

A unary operator must not have explicit parameter lists even if they are empty. A unary operator is a method named "unary_`op`" where `op` is one of `+`, `-`, `!`, or `~`.

Progra... Wildca...