Proposals for NetRexx [2008-03-23]

In this document I describe eight proposed areas of enhancements to the NetRexx language: piece-wise methods definitions, privileged objects, enhanced type system, short-circuit logical operators, assertions, enhanced array support, default implementations in interfaces, and delegates. My intention in making those proposals is to enhance the readability, expressivity, and descriptiveness of NetRexx, so that the programs we create can be easier to write and maintain. The syntax and semantics are subject to change. Please send comments and suggestions to Martin Lafaix, lafaix@online.fr.

Contents

1. Purpose

Given that you are reading this document, I assume that you want to know how to enhance NetRexx. Given that you have that desire, I assume that you already know about NetRexx.

My purpose in this document is double: I want to propose enhancements to the NetRexx language, and I would like to have your opinions and comments and suggestions on those enhancements (or on other enhancements you may desire).

I have been using NetRexx since its first public release more than three years ago. I like the language very much, but I have found some of its features to be lacking, and I miss several features present in other languages. Although I tested my (poor :-) lobbyist skills in the past years by making a few informal suggestions, I think the time has come for a more reasoned approach---after all, the NetRexx language definition has been stable for quite some time, and hence demands consideration.

My intention is to address eight areas in which I consider enhancements are most needed. Those areas are, in no specific order:

- methods definitions
- privileged objects
- the type system
- logical operators
- assertions
- arrays
- default implementation of methods in interfaces
- delegates

Each area is the subject of its own section, in which I propose one or more enhancements. The enhancements are independents: adding one of them does not require the addition of another (but it does not mean they do not relate, in that an enhancement may improve the functionality of another).

By making those proposals, I try to enhance NetRexx's consistency, readability, expressivity and descriptiveness. In other words, I suggest implementing those proposals so that we can create programs more effectively.

But without some feedback from other NetRexx users---namely you---my proposals are of little interest. There is no point in adding features to the NetRexx language if they are only used by one person---namely me. That is the reason why your opinions and comments are essential. Without your participation, this document is of no use. So please express yourself, and feel free to make your own proposals.

In this document, I am only considering the NetRexx language. I am not taking into account a particular implementation or a particular platform an implementation may rely on. Although the only NetRexx implementation I know of relies on Java, the language by itself is not Java dependent. I have tried not to add such dependencies in my proposals.

It also means that I am making no proposals related to a particular implementation. You will see no performances or compile-time--optimization proposals. Such proposals are not related to the NetRexx language per se, hence they do not fit the purpose of this document. (But it does not imply a given proposal may not open the road to a more efficient implementation---indeed, some proposals do allow for an efficient implementation.)

In this document, I use various notations such as changes of font for clarity and for consistency with the NetRexx language definition. Within the text, I use a sans-serif bold font to indicate **keywords**, and the same font in italic to indicate *technical terms*. I use an italic font to indicate a reference to a *technical term defined elsewhere* or a *word* in a syntax diagram that names a segment of syntax.

Similarly, in the syntax diagrams, I use a bold font for words (symbols) that also denote **keywords** or sub-keywords, and italic for words (such as *expression*) that denote a token or collection of tokens defined elsewhere. The brackets [and] delimit optional (and possibly alternative) parts of the instruction. An ellipsis (...) following a bracket indicates that the bracketed part of the clause may optionally be repeated.

Occasionally in syntax diagrams brackets are "real" (that is, a bracket is required in the syntax; it is not marking an optional part). I enclose such brackets in single quotes, thus: '**[**' or '**]**'. This may also occur for ellipsis.

In some sections, I may provide additional information on some points. This additional information is placed in a side bar like this one. Such additional information complements the proposal, by giving references or explanations.

When the symbol

appears at the end of a section, it warns of a "dangerous bend" in the train of thought; do not read the end of the section unless you need to. (By "dangerous bend" I mean talk about unresolved or undecided issues.)

2. Piece-Wise method definitions

NetRexx supports multiple definitions for a method, as long as those definitions have different signatures. I suggest extending this behavior by also allowing the use of constants, *patterns*, and *predicates* in place of arguments, so that we can provide multiple definitions for a method for specific arguments values.

2.1. Constants

It is not infrequent for us to write methods whose bodies consist of either a **select** block or nested **if**s instructions. A typical example is the factorial function:

```
method fact(n)
  if n = 0 then
    return 1
  else
  if n > 0 then
    return n * fact(n-1)
  else
    -- invalid arg
```

When the method is small, the code is readable and the intent not too obscure. But as soon as the number of cases increases, or if the code becomes large, our intent is less clear. We have to look at the end of the method to be sure the **select** or **if**s are not followed by trailing code. We no longer immediately know that the body of the method is in fact a series of independent parts.

So I suggest adding piece-wise method definitions to NetRexx. Instead of having to create a big **select** or nested **if**s block, we simply enumerate the various partial definitions:

Example 2.1:

```
method fact(0)
  return 1

method fact(n | n > 0)                  -- or fact(n when n > 0)
  return n * fact(n-1)
```

In general a *piece-wise definition* of a method consists of two or more parts. Each part gives a piece of the entire definition. As with a **select** block, the first matching definition is used and it is an error to either omit a case or redefine an already handled case.

All partial definitions for a given signature should be grouped together to allow line number conservation for implementations relying on an external compiler, such as javac. A piece-wise definition ends when one of the following elements is encountered:

- a new method definition (i.e., a **method** instruction with a different signature or a different name)
- a **class** instruction
- the end of the program

The type of a given argument can be specified. For example, if the we want to force the argument of the preceding `fact` method to be an `int`, changing `0` to `int 0` and `n | n > 0` to `n = int | n > 0` will do it.

If exceptions are declared or signaled by pieces, their union is used for the signature. (If one of those exceptions is a subclass of another specified exception, it is omitted.)

This construction makes our intent clear. It also lets the compiler implement the argument selection efficiently. More important, this construction opens the road to efficient and simple argument deconstruction.

2.2. Patterns

As of now NetRexx only provides deconstruction for strings via the **parse** clause. Piece-wise definitions allow us to use deconstruction directly in argument specification:

Example 2.2:

```
method hms2s(hh ':' mm ':' ss)          -- instead of:
  return hh*3600 + mm*60 + ss           --   method hms2s(dummy)
.                                       --     parse dummy hh ':' mm ':' ss
.                                       --     return hh*3600 + mm*60 + ss
.
say hms2s('21:37:07')
```

As shown in Example 2.2, by using a pattern in the argument specification we make our intent clear. The code is concise yet readable. We no longer have to create unnecessary variables. We simply expose our intent.

If array deconstruction is added to NetRexx (see Section 7.2), an array deconstruction pattern would also be allowed in argument specification.

This use of patterns in argument specifications differs from *pattern matching* as found in for example Standard ML. A pattern in NetRexx always matches. In Example 2.2, calling `hms2s('foo')` will produces a `NumberFormatException` at runtime with the reference implementation. It will *not* signal an unmatched specification. To achieve this, we have to use predicates (see Section 2.3).

2.3. Predicates

The use of constants in piece-wise definitions allows us to discriminate among discrete conditions, but we cannot specify intervals or nondiscrete conditions. This is too restrictive.

I suggest allowing predicates in argument specification, as shown in Example 2.1. A predicate is an optional construction following an expression (the vertical bar "`|`" is pronounced "such that" in this context):

```
['|' expression ]
```

A predicate can only refer to visible objects: (1) arguments specified on the left of the predicate, (2) members of the object the method being defined is part of, and (3) visible static members of any class.

We can use as many predicates as we wish in a given method definition. The definition will match if and only if the predicates (i.e., the expressions following the "`|`") evaluate to `1`.

examples:

```
method foo(x | x + y > 0, y)          -- invalid, y not visible yet
method foo(x, y | x + y > 0)          -- correct
method foo(x | x > 0, y | y > 0)
method bar(x | Math.sin(x) < 0)
method baz(x | x.left(3) = 'foo')
```

We can also use a predicate after the argument list in a function declaration, so that we can write code such as `method foo(x, y) | x + y > 0`. We can use a predicate even when no argument exists.

Predicates enhance piece-wise definitions by allowing us to specify the conditions we desire. We can write programs effectively, and our intent is clear.

Using a constant as an argument can be seen as a special case of a predicate: `"foo"` is equivalent to `dummy | dummy = "foo"`.

"`|`" is maybe not the most appropriate notation for introducing predicates (despite that it is not ambiguous and that such overloading cases are already presents in NetRexx---consider for example `x = aBoolean = anotherBoolean`). A textual keyword such as **when** or **suchthat** may be less clumsy. If a textual keyword is used, it would be wise to handle it like the **then** keyword in **if** instructions, so that it is not mistaken for a pattern element.

2.4. Summary

By extending the current method definition mechanism, we enhance the descriptiveness of NetRexx. The use of patterns and predicates makes our intent more obvious. The programs we write are closer to what we think.

Additionally, an automatic documentation tool---such as *nrxdoc*---could be designed to use this new information. This enhancement would help to make the documentation clearer and more accurate, and would reduce the risk of it getting out of sync. Other tools such as debuggers, profilers, and so on could also use this information.

There is a need for a placeholder in parameter list (so that we do not have to name an argument we are not using). Two visually appealing candidates are "_" and "`...`". The first candidate is unfortunately a valid variable name. Hence using it for this purpose would break backward compatibility. The other candidate can cause comprehension problems in that an ellipsis usually stands for a variable number of elements. Using either a single "`.`" or nothing (as in REXX) is a possible solution, but I do not find it readable enough in the context of a parameter list. Suggestions welcome!

Having to repeat the method signature can be annoying. Devising a way to avoid that would be interesting. Ideas welcome!

3. Privileged objects

When writing a program, it is not uncommon for us to call methods or set and or query properties of a given object repeatedly. As shown in Example 3.1, this leads to verbose and error-prone code:

Example 3.1:

```
say rectangle.x rectangle.y rectangle.width rectangle.height
```

We need a way to denote a *privileged* object, so that we can access it more effectively. I suggest adding an optional **with** keyword to all block structure instructions (namely, **class**, **do**, **loop**, **method**, and **select**):

[**with** *term*]

term has either to be visible at the time of its declaration or to be the special `this` term. If `this` is used, the privileged object will be the currently running instance.

Inside such a qualified structure, we can refer to the privileged object's members by omitting the privileged object name. This eluding is only allowed at the beginning of a term. This usage remains possible within non--**with**-qualified enclosed structures.

Example 3.2:

```
do with rectangle
  say .x .y .width .height
end
```

The *term* is evaluated each time the **with**-qualified block body is run. It means that, in Example 3.2, `rectangle` is evaluated once. In Example 3.3 the term is evaluated once per iteration:

Example 3.3:

```
loop i = 0 for foo.length                -- or equivalently:
  do with foo[i]                         --
    say .bar                             -- loop i = 0 for foo.length with foo[i]
  end                                    --   say .bar
end                                      -- end
```

The privileged object that may have been defined for a class is not inherited by its possible subclasses. If no **with** keyword is specified for a top-level **class** instruction, **with** `this` is assumed.

This construction preserves backward compatibility. It helps us to write concise yet readable code. It does not add much complexity to NetRexx.

The **with** mechanism shares many similarities with the **uses** keyword found in **class** instructions. The purpose is different, but the underlying concepts are alike.

They both allow for concise notation, the two mechanisms affect only the syntax of terms in the current class, and the subclasses do not inherit them. Also, although the **uses** instruction is currently only attachable to **class** instructions, we can imagine extending the use of **uses** keyword to other block structures (the limiting factor here being that it might be of little interest).

It may be interesting to restrict the type of *term*. Built-in types such as `int` or `double` are of no interest. (A primitive type is not an `Object`---hence calling methods is not valid.)

4. Enhanced type system

The type system is an essential part of a programming language. NetRexx offers a simple yet convenient type system. It features single inheritance plus interfaces and a limited type inference algorithm (the first use of a untyped variable determines its type).

Although convenient, I find this scheme to be too simple in a nonnegligible number of cases: We cannot express some known type restrictions at compile time, we have to write repetitive code to handle similar types, and we have to explicitly specify the type of some variables.

4.1. Bounded types

The currently provided type system does not always allow us to express type restrictions we would like to enforce. For example, let us assume we want to code a method that requires an argument that is both a `Component` and `Cloneable`.

Due to the `Component` class being both out of our control and not implementing the `Cloneable` interface, we cannot enforce that restriction in our method definition. Instead, we have to rely on a runtime check:

Example 4.1:

```
method foo(c = Component)                -- or  ... c = Cloneable
  if \ (c <= Cloneable) then             -- and ... c <= Component
    signal IllegalArgumentException(...)
  ...
```

This implies some errors we *know* could occur will not be caught at compile time. We cannot circumvent this problem. Defining a new class extending `Component` and implementing `Cloneable` is of no use as existing subclasses of `Component` would remain unaffected. To resolve this deficiency, I suggest adding *bounded type*s to NetRexx. A bounded type is an intersection of types.

A bounded type can appear anywhere a type is allowed. The object is assumed to have all types constituting the intersection. The following notation is used to denote bounded types:

type **&** *type* [**&** *type*]...

where *type* is a type

At most one *type* can be a descendant of `Object`. The other types must be interfaces. (If we were to allow more than one noninterface type, this would be either redundant, one type being a subclass of another, or impossible---multiple inheritance does not exist in NetRexx, and hence an object cannot belong to two nonrelated classes.)

A bounded type usually needs to be parenthesized if it is followed by an expression, as the "`&`" operator has a lower precedence than the blank operator.

When an object's type is a bounded type, it means we can access all visible members of any of the union's constituents. The following code requires no explicit casts:

```
method bar(c = ActionListener & WindowListener)
  ...
  c.actionPerformed(...)
  ...
  c.windowClosed(...)
  ...
```

By using bounded types, we could rewrite Example 4.1 as follows:

Example 4.2:

```
method foo(o = Component & Cloneable)   -- or Cloneable & Component
  ...
```

Example 4.2 is simpler and safer. If we mistakenly pass an incorrect argument, the error will be caught at compile time.

Bounded types enhance the safety of our code, and hence our productivity. In fact, due to us also not having to explicitly specify a cast, the code we write is more concise and more readable. The notation induces no incompatibility with existing code.

The bounded type information could be lost in a Java class file. Although we could save it in an attribute, a nonaware compiler would fail to see it. By mapping the bounded type to its erasure (as defined in the Generics for Java draft), we would ensure compatibility with other existing Java compilers. The advanced knowledge would be lost, but that already is the case right now, so...

The *Adding Generics to the Java Programming Language* draft (cf. http://jcp.org/aboutJava/communityprocess/review/jsr014/), in its Translation section, defines a simple way to implement bounded types in a backward compatible way. If we are to add bounded types to NetRexx, I think we should follow this specification.

4.2. Types synonyms

The Java Class Libraries contain many meaningfully-named classes. Alas, although being meaningful, those names can be long to type and to read. Even more, to be able to resolve name conflicts, we have to use complete class name from time to time (e.g., is `List` the AWT one or the collection one? or is it a call to a `list()` method?).

I hence suggest adding simple *types synonyms* to NetRexx.

NetRexx's types synonyms are attached to a block structure (i.e., a **class**, **do**, **loop**, **method**, or **select** instruction). They are active from their definition point to the end of the defining structure. It is not legal to redefine a synonym previously specified in an enclosing scope. A synonym can use a previously-defined one, but it cannot use one defined after it. In other words, recursive types synonyms are *not* allowed.

Types synonyms are defined via a new clause:

name == *term*

The symbol naming the type synonym, *name*, cannot begin with a digit (0-9). The synonym substitution is by default case-insensitive (but this case sensitivity can be modified by the **strictcase** option).

The body of the synonym, *term*, must be a class name (simple or compound).

The substitution only occurs for symbols. It does not occurs in literal constants. It can occur in compound terms only if the symbol is at the starting position. It is a straight substitution. No parameters are allowed. No recursive substitution is attempted (in other words, the result of a symbol's substitution is not analyzed for more possible substitutions).

Example 4.3:

```
class foo extends Component -
          implements MouseListener

  AWTList == java.awt.List
  ME      == MouseEvent

  method foo
    ...
    aList = AWTList()
    ...
  method mouseEntered(e = ME)
    ...
  method mouseExited(e = ME)
    ...
```

The types synonyms that may have been defined for a class are not inherited by its possible subclasses.

Types synonyms allow us to write concise code in places no other mechanisms would. By writing concise yet clear code, we enhance our productivity and the readability of our code.

4.3. Views

We do not always control the classes we use. We often rely on third-parties libraries. This helps us to shorten our development costs. But there is one problem that may arise with third-parties libraries: discrepancies may exist between two similar classes we use. By discrepancies I mean that two classes that may provide an essentially identical service, due to having been developed by two different parties, may not have the same interface.

It implies that we have to duplicate our code if we for example want to develop a method that may use those two (or more) similar services. This required duplication of code is costly. It reduces the readability and maintainability of the code we write. Our code becomes error prone.

So I suggest adding *view*s to NetRexx. A view is a way to instruct the system that two types are in essence identical (at least for the use we have of them). A view is defined as follows:

> **view** *type1* **as** *type2* **;**
>
> where *type1* and *type2* are types

If needed, this instruction is followed by one or more methods definitions. Those methods definitions are the necessary glue that is needed to make an instance of *type1* acts as an instance of *type2*. We do not have to provide a definition for all *type2*'s methods---just for the ones we use. When convenient, private methods are allowed. Such private methods cannot be called directly from the rest of the program. They are only callable from the view's methods.

A view is visible (and hence active) from its point of definition up to the end of the enclosing structure (either a class or a program). The definition of a view is ended by a **view** instruction or a **class** instruction. A **view** instruction ends the definition of a possible preceding **class** instruction, except if the view definition is followed by minor-class definitions.

A view is not an object per se. It does not exist at run time. When we define a view, we specify that every method accepting an instance of *type2* must also accept an instance of *type1* as an alternative. In other words, we multiply the number of methods.

type1 and or *type2* may be simple or bounded types. They can be interfaces or abstract classes. There is no need for *type1* and *type2* to be two related types---that is, one of them does not have to be a subclass of the other. Although *type1* can be a primitive type, *type2* cannot.

In Example 4.3, we have a simple class that defines a static method `myMethod`. Due to the specified view, a new method is generated:

Example 4.4:

```
-- MyClass.nrx
view Component as Container
  method paintComponents(g = Graphics)

class MyClass
  method myMethod(c = Container) static
    ...
    c.paintComponents(...)
    ...


-- MyTest.nrx
MyClass.myMethod(aContainer)            -- OK
MyClass.myMethod(aComponent)            -- OK too, due to the view
MyClass.myMethod(anObject)              -- Error!
```

Views hide the differences between two types. From our point of view, they become identical. Hence, views simplify our work, and enhance our productivity.

In other words, views allow us to modify hierarchies we have no control upon. The source code is no longer needed if we want to make a modification to a base class.

4.4. Indexed Types

Example 4.5:

```
+++ 2.1

+++ Example 1  Indexed types

Vector[String]
Seq[Seq[A]]
Seq[String].Zipper[Integer]
Collection[Integer]
```

```
Pair[String, String]

-- Vector[int] -- illegal, primitive types cannot be arguments
-- Pair[String] -- illegal, not enough arguments
-- Pair[String, String, String] -- illegal, too many arguments

+++ 2.2

x = Vector[String]()
y = Vector[Integer]()

return x.getClass = y.getClass -- yields true

+++ Exemple 2  Mutually recursive type variable bounds

class ConvertibleTo[A] interface forall A

  method convert returns A


--class ReprChange[A = ConvertibleTo[B], B = ConvertibleTo[A]]
class ReprChange[A, B] forall A = ConvertibleTo[B], B = ConvertibleTo[A]

  a = A

  method set(x = B)
    a = x.convert

  method get returns B
    return a.convert

+++ Example 3  Nested indexed class declarations

class Seq[A] forall A

  head = A
  tail = Seq[A]

  method Seq()
    this(null, null)

  method Seq(hd = A, tl = Seq[A])
    head = hd
    tail = tl

  method isEmpty returns Boolean
    return tail = null

class Seq[A].Zipper[B] dependent forall B

  method zip(that = Seq[B]) returns Seq[Pair[A, B]]
    if this.isEmpty, that.isEmpty then
      return Seq[Pair[A, B]]()
    else
      return Seq[Pair[A, B]](Pair[A, B](this.head, that.head), -
                             this.tail.zip(that.tail))

  -- alternative implementation, if we were to use the same
  -- type inference algorithm for methods and constructors

  method zip(that = Seq[B]) returns Seq[Pair[A, B]]
    if this.isEmpty, that.isEmpty then
      return Seq()
    else
      return Seq(Pair(this.head, that.head), -
                 this.tail.zip(that.tail))

class Client

  strs = Seq[String]("a", Seq[String]("b", Seq[String]()))

  nums = Seq[Number](Integer(1), Seq[Number](Double(1.5), -
                                             Seq[Number]()))

  -- alternative

  strs = Seq[String] Seq("a", Seq("b", Seq()))
  nums = Seq[Number] Seq(Integer(1), Seq(Double(1.5), Seq()))

  zipper = strs.Zipper[Number]()
  combined = zipper.zip(nums)
```

```
+++ 2.4

class B implements I[Integer]
class C extends B implements I[String]

+++ 2.5

+++ Example 4  Raw types

class Cell[A] forall A

  value = A

  method Cell(v = A)
    value = v

  method get returns A
    return value

  method set(v = A)
    value = v

x = Cell Cell[String]("abc")
x.value
x.get
x.set("def") ++ deprecated

+++ 3.1

+++ Example 5  Polymorphic methods

  method swap(a = Elem[], i, j) static forall Elem
    temp = a[i]; a[i] = a[j]; a[j] = temp

  method sort(a = Elem[]) forall Elem = Comparable[Elem]
    loop i = 0 for a.length
      loop j = 0 to i-1
        if a[j].compareTo(a[i]) < 0 then
          swap(a, i, j)
      end
    end


class Seq[A] forall A

  method zip(that = Seq[B]) returns Seq[Pair[A, B]] forall B
    if this.isEmpty, that.isEmpty then
      return Seq[Pair[A, B]]()
    else
      return Seq[Pair[A, B]](Pair[A, B](this.head, that.head), -
                             this.tail.zip(that.tail))

+++ 3.2

+++ Example 6  valid now (was invalid according to JLS)

class C implements Cloneable

  method copy returns C
    return C clone


class D extends C implements Cloneable

  method copy returns D
    return D clone

+++ 4

+++ Example 7  Polymorphic signals clauses

class PrivilegedExceptionAction[E] interface forall E = Exception

  method run signals E


class AccessController

  method doPrivileged(action = PrivilegedExceptionAction[E]) static signals E forall E = Exceptio
    ...
```

```
class Test

  method main(args = String[]) static
    do
      AccessController.doPrivileged(PEAction())
    catch FileNotFoundException f
      ...
    end

class Test.PEAaction implements PrivilegedExceptionAction[FileNotFoundException]

  method run signals FileNotFoundException
    ...

+++ 5.1

+++ Example 8  Class instance creation expressions

Vector[String]()

Pair[Seq[Integer], Seq[String]](Seq[Integer](Integer(0), Seq[Integer]()), -
                                Seq[String]("abc", Seq[String]()))

-- alternative

Pair[Seq[Integer], Seq[String]] Pair(Seq(Integer(0), Seq()), -
                                     Seq("abc", Seq()))
+++ 5.2

+++ Example 9  Array creation expressions

Vector[String][n]
Seq[Character][10][20][]

+++ 5.3

+++ Example 10  Assume the declarations

class Dictionary[A, B] extends Object forall A, B
  ...

class Hashtable[A, B] extends Dictionary[A, B] forall A, B
  ...

d = Dictionary[String, Integer]
o = Object

Hashtable[String, Integer] d ++ legal, has type Hashtable[String, Integer]
Hashtable o                  ++ legal, has type Hashtable

Hashtable[Float, Double] d   ++ illegal, not a subtype
Hashtable[String, Integer] o ++ illegal, not unique subtype

+++ 5.4

+++ Example 11  Type comparisons

class Seq[A] implements List[A] forall A

  method isSeq(x = List[A]) static returns boolean
    return x <= Seq[A]

  method isSeq(x = Object) static returns boolean
    return x <= Seq

  method isSeqArray(x = Object) static returns boolean
    return x <= Seq[]

+++ Example 12  Type comparisons and type casts with type constructors

class Pair[A, B] forall A, B

  fst = A
  snd = B

  method equals(other = Object) returns boolean
    return other <= Pair & -;
           equals(fst, (Pair other).fst) & -;
           equals(snd, (Pair other).snd)
  method equals(x = Object, y = Object) private returns boolean
    -- x = null &&& y = null ||| x \= null &&& x.equals(y);
    if x = null & y = null then;
      return 1
    if x \= null then
      return x.equals(y)
    return 0

+++ 5.5

+++ Example 13  Polymorphic method calls (see Example 5)

swap(ints, 1, 3)
sort(strings)
strings.zip(ints)

+++ 5.6

+++ Example 14  Type parameter inference

  method nil static returns Seq[A] forall A
    return Seq[A]()

  method cons(x = A, xs = Seq[A]) static returns Seq[A] forall A
    return Seq[A](x, xs)

  cons("abc", nil)                       -- of type Seq[String]
  cons(IOException(), cons(Error(), nil)) -- of type Seq[Throwable]
  nil                                    -- of type Seq[*]
  cons(null, nil)                        -- of type Seq[*]

+++ Example 15  An unsafe situation for type parameter inference

  method duplicate(x = A) returns Pair[A, A]
    return Pair[A, A](x, x)

  method crackIt(p = Pair[Seq[String], Seq[Integer]])
    p.fst.head = "hello"
    i = p.snd.head

  crackIt(duplicate(cons(null, nil))) -- illegal !

+++ 6.2

+++ Example 16  Bridge methods

class C[A] forall A

  method id(x = A) returns A abstract

class D extends C[String]

  method id(x = String) returns String
    return x

-- translates to

class C

  method id(x = Object) returns Object abstract

class D extends C

  method id(x = String) returns String
    return x

  method id(x = Object) returns Object
    return id(String x)

+++ Example 17  Bridge methods with the same parameters as normal methods

class C[A] forall A

  method next returns A abstract

class D extends C[String]

  method next returns String
    return ""
```

```
-- translates to

class C

  method next returns Object abstract

class D extends C

  method next/*1*/ returns String
    return ""

  method next/*2*/ returns Object
    return next/*1*/

+++ Example 18  Overriding with covariant return types

class C

  method dup returns C
    ...

class D extends C

  method dup returns D
    ...

-- translates to

class C

  method dup returns C
    ...

class D extends C

  method dup/*1*/ returns D
    ...

  method dup/*2*/ returns C
    return dup/*1*/

+++ Example 19  Name clash excluded by Rule 2

class C[A] forall A

  method id(x = A) returns A

class D extends C[String]

  method id(x = Object) returns Object
    ...

-- illegal since C.id and D.id have the same type erasure, yet D.id does
-- not override C.id.

+++ Example 20  Name clash excluded by Rule 3

class C[A] forall A

  method id(x = A) returns A

class I[A] interface forall A

  method id(x = A) returns A

class D extends C[String] implements I[Integer]

  method id(x = String) returns String
    ...

  method id(x = Integer) returns Integer
    ...

-- illegal since C.id and I.id have the same type erasure yet there is no
-- single method in D that indirectly overrides C.id and implements I.id.

+++ 6.3

class Cell[A] forall A
```

```
    value = A

    method getValue returns A

  ...

  method f(cell = Cell[String]) returns String
    return cell.value

  -- the return statement needs to be translated to

    return String cell.value

  -- in the previous context

  x = cell.getValue

  -- needs to be translated to

  x = String cell.getValue

  +++ other

    method foo(a = Component & Cloneable);
    method foo(a = (Component & Cloneable) null);
    if x <= Object & Cloneable then +++ illegal;
    if x <= (Object & Cloneable) then;
    method foo(a = ActionListener & WindowListener) returns ActionListener & WindowListener;
    method foo(a = A) returns A forall A = ActionListener & WindowListener;
```

4.5. Type inference

The currently used *type inference algorithm* rely on the type of the variable at its first assignment (when we do not define this type explicitly). Similarly, the type of an array initializer depends of the type of its first element.

Although useful, this kind of type inference forces us to specify the type of a variable if its first assignment is not of the desired type (i.e., the variable is for example assigned an object whose type is a subclass of the intended type). Therefore, I suggest using a more elegant algorithm for local variables and array initializers type inference. The type inference algorithm for properties and methods is not modified, to maintain compatibility and predictability.

Instead of relying on the first assignment of a local variable, all uses are considered. The assigned type is either (1) the type specified at declaration time or (2) the common compatible type enclosing all the variable uses. If no compatible type is found, a compile-time error is issued.

If the type determination is not possible due to recursion or ambiguity, a compile-time error is issued. We solve this by explicitly declaring the local variable to be of a given type. When we do not specify the local variable type, the inferred type is the deepest possible one in the class hierarchy.

Similarly, for array initializers, the type of all elements is considered. If the inferred type is not the one we want, we can declare the type of an element of the array explicitly, as in `[int 1, 2, 3]`. Such an explicit declaration overrides the inferred type. If the type of an element is incompatible with this explicit declaration, a compile-time error is issued.

This proposal preserves backward compatibility. Previously valid programs remain unaffected.

Similar type inference algorithms [Hindley; Milner] exist for languages such as Standard ML. Although this kind of algorithm is not trivial and can theoretically lead to doubly-exponential execution time, this is (1) a known technique and (2) the execution time is nonexponential for *all* practical cases. (That is, whereas the algorithm is indeed doubly-exponential, there exists no finite program that exhibits this double-exponentiality---at least it is the case for Standard ML and other affiliated languages. It remains to be seen for NetRexx, but I foresee no reason...)

Trying to generalize the type inference algorithm to handle the methods parameters types would be interesting. I have not researched it enough yet, so I am not sure it would provide useful results. I plan to write a NetRexx front end to test that, but my expected release date is 2050 :-)

I would also like to typecheck untyped empty arrays (i.e., `[]`) whenever the typing is not ambiguous. If only one method matches, then I would like the code to compile. If more than one method match, then I would like a compile-time error. It implies a previously compilable code could become incorrect, but I do not find this behavior worse than the current one concerning automatic conversions (i.e., `foo(1)` will call `foo(long)` if it is the only available method, but if we later add a `foo(byte)` this new method will be called if we recompile our code).

4.6. Summary

By making the type system stricter and smarter, we would make NetRexx more versatile and less constraining. Bounded types mean we can enforce stricter control at compile time, and type inference means we program more effectively.

Views prevent us from writing repetitive code while maintaining type safety. The code we write is more readable and our productivity increases.

With such an enhanced type system, NetRexx would be more expressive, easier to program in, and safer.

5. Short-circuit logical operators

The logical operators currently provided by NetRexx evaluate their arguments fully. For example, if we want to check whether a string is either null or empty, the following code is needed:

Example 5.1:

```
if str = null then
  -- action1
else
if str = "" then
  -- action1 again
else
  -- action2
```

We cannot use a piece of code such as `if str = null | str = "" then ...` ---this code is invalid in that it will cause an exception if *str* is `null` (due to the second check).

The duplication of action1 is inevitable, and our intent is not obvious. If the logical expression we want to test is more complex and if it contains incompatible parts or possibly very long computations, the code we have to write quickly becomes nonmaintainable. This uncalled-for complexity in our code is caused by the need for the logical operators to evaluate all their arguments, even when doing so is unnecessary.

I suggest adding two *short-circuit* logical dyadic operators to NetRexx. A short-circuit logical dyadic operator is an operator that evaluates its right-hand--side argument only if this evaluation is necessary. (Such a partial evaluation makes sense for logical "and" and "or" operators. If the left-hand--side argument of an "and" operator is `0`, the result is already known---it will be `0`. Similarly, if the left-hand--side argument of an "or" operator is `1`, the result is also known---it will be `1`. On the opposite, it does not make sense for the "xor" operator---in this case, the values of the two arguments are always needed.)

I propose the "`|||`" and "`&&&`" notations for the short-circuit "or" and "and" operators, respectively.

With this proposal, we could rephrase Example 5.1 as:

```
if str = null ||| str = "" then
  -- action1
else
  -- action2
```

By using a short-circuit logical operator, we no longer have to duplicate the code for action1. Hence the possibility of introducing an error during the duplication is removed. The code we write is shorter, clearer, and less error-prone.

Because those operators are not currently valid NetRexx constructs, this proposal adds no incompatibility with current code. The risk of introducing those operators by mistake in existing code and producing a valid program (but whose behavior could lead to surprises) is also quite low (not more so than for example a mistake between "`&`" and "`&&`").

Implementing those features by specifying a compiler switch or an **options** option can cause problems when using an old version of a compiler and would prevent the simultaneous usage of both short-circuit and non--short-circuit logical operators in a program.

Short-circuit logical operators are not bitwise operators. Although we could possibly extend their definitions to support integer arguments---all `0` for an "and" left-hand side means `0` for the result, all `1` for an "or" left-hand side means all `1` for the result---it would cause problems in the general case. The "all `0`" case is easy, but the "all `1`" one is trickier. It implies we know the length in bits of both arguments---which means that, in the general case (e.g., a `Rexx` integer), we have to compute the value of both arguments. Even though those problems do not arise if the arguments are of a primitive integer type of a known size, I think it is better not to extend the behavior of short-circuit logical operators to support bitwise operations than to introduce inconsistencies. As the proposed operators use their own notation, we can do so without difficulties.

NetRexx 2 introduces a notation that provides, in some limited (but useful) cases, an equivalent for the short-circuit "or" operator: the "," (comma) operator in **if** and **when** instructions. This notation can not be used in more general contexts, and does nothing for the "and" short-circuit operator, non-trivial logical expressions, and logical expressions used in method calls, though.

6. Assertions

NetRexx provides a **trace** instruction. This instruction helps immensely during code debugging phases, as we can enable and or disable it via a compiler switch, with no source code modification.

A complementary technique is the usage of *assertions*. This classic technique introduces the notion of *pre-* and *post-conditions*, as well as *class invariants*:

- An assertion is a checkpoint placed in the body of a method. It verifies the validity of a condition at this point.
- A precondition applies to a method. It is a check that is made just before running the body of the method.
- A post-condition also applies to a method. It is a check that is made just before returning control to the program or the point of invocation. If the method returns a value, the check is done after the returned value is computed, but before actually returning the value.
- A class invariant applies to an object. It is a property of the object that is true after the construction of the object and between invocations of the object's **public** (or **shared**) members (but it does not have to apply during the execution of the object's methods).

We can emulate assertions by decorating the code, but doing so has one drawback: there is no way we can disable those emulated instructions without modifying the code. Although we could permanently enable the assertions, this could induce a severe performance penalty and, instead of feeling free to exploit those features, we would refrain ourself in their use. (Another possibility is to maintain two source trees---one for testing, one for production---but this requires adequate tools.)

I hence suggest adding four instructions to NetRexx: **assert**, **ensure**, **invariant**, and **require**, for respectively assertions, post-conditions, invariants, and preconditions. Those instructions are followed by a logical expression:

assert [**label** *name*] *expression* **;**

ensure [**label** *name*] *expression* **;**

invariant [**label** *name*] *expression* **;**

require [**label** *name*] *expression* **;**

If the logical expression is true, the assertion is said to succeed. If it is false, the assertion is said to fail. When an assertion fails, the execution is interrupted and the offending assertion is displayed (with module name and line). A given implementation may decide to display more information (such as the call stack).

The **invariant** instruction must appear at the begining of a class, before any property or method. There can be more than one **invariant** instruction in a class, in which case they must all apply (in other words, this notation is equivalent to linking the expressions with a logical "and"). The **ensure** and **require** instructions can appear anywhere in the body of a method, provided they are not in a block instruction (**do**, **if**, **loop**, or **select**). The **assert** instruction can appear anywhere in the body of a method.

There can be as many assertions as needed for any given class or method. Although a unique **assert** instruction could fulfill the role of the three other instructions, using distinct names allows us to enable and or disable them selectively. It also provides immediate visual differentiation.

With the addition of some more option words for the **options** instruction, this allows for a convenient and powerful usage of assertions in NetRexx. I propose the following option words: **assert**, **assert0**, **assert1**, **assert2**, **ensure**, **invariant**, **require**, and their "no-" variants (except for the numbered "asserts"). **assert0** disables *all* assertions (that is, it is equivalent to **noassert noensure noinvariant norequire**). **assert1** means all output goes to stdout, **assert2** to stderr (the default).

Example 6.1:

```
class Buffer
  invariant in - out >= 0 & in - out <= buffer.length

  ...

  method addElement(o = Object)
    require o \= null & \ isFull
    ensure \ isEmpty & contains(o)
    ...
```

Assertions, when directly supported by the language, are extremely useful. We can write correct and safe code quickly. The inescapable debugging phase is immensely simplified. The ability to enable or disable them at will at compile time is essential: it allows us to use assertions without fear.

The class invariants and the pre- and post-conditions are part of the class definition and hence provide documentation. An automatic documentation extraction tool---such as nrxdoc---can include them in the documents it produces.

A presentation of the Design by Contract notion (i.e., the use of class invariants and pre- and post-conditions as part of the class definition) is available at http://www.eiffel.com/doc/manuals/technology/contract/page.html.

Proposal: A Simple Assertion Facility For the Java(tm) Programming Language.

What remains to be discussed is what option is enabled by default (none? or just pre- and post-conditions?), and whether assertions are implemented via exceptions or something else. Using exceptions (as in Eiffel) means we can catch them, but is this desirable?

Another pending question is how---if any---to implement and or handle the notion of previous state in post-conditions. I do find the Jass way interesting (see http://semantik.informatik.uni-oldenburg.de/~jass).

I have removed the explicit discussion of loop variants and invariants. I am not sure whether I should re-introduce it.

In case of piece-wise method definitions, the assertions specified for a piece only apply to it. I do not know how to enforce coherence between the post-conditions, though.

I do not know how to enforce coherence for pre- and post-conditions in inherited methods. (The theory says the redefined preconditions should be weaker and the redefined post-conditions stricter.)

Thinking of pre- and post-conditions, what about restricting their possible position in a method's body? (Forcing them to be at the beginning may enhance readability.)

7. Arrays

Arrays, although being an integral part of NetRexx, are in many ways second-class citizens. We can declare them, create them, query their lengths, and query and set their elements. But those are the only available built-in operations---there is no higher-level operations. We cannot easily concatenate two type-compatible arrays, or even walk through the elements of an array without having to take care of the gory implementation details.

The lack of higher-level operations for arrays does not prevent us from writing code to circumvent it, but this explicit (and possibly repetitive) coding favors errors and reduces readability. Hence this section, in which I propose more support and integration for arrays.

7.1. Concatenation

NetRexx allows us to define a literal array easily via array initializers. But the other expectable way to create an array, that is, concatenation, is not available. Or, more accurately, we cannot concatenate two arrays in most cases. If we are for example using `char` arrays, we *can* concatenate them. This partial support is inconsistent and confusing.

Therefore, I suggest supporting concatenation for any two type-compatible arrays. There is no need for a new operator: we can reuse the `||` operator for concatenating two arrays.

```
foo = [1, 2, 3]
bar = [4, 5, 6, 7]
foobar = foo || bar

-- instead of
--
-- foobar = Rexx[foo.length+bar.length]
-- loop i = 0 for foo.length
--   foobar[i] = foo[i]
-- end
-- j = 0
-- loop i = foo.length for bar.length
--   foobar[i] = bar[j]
--   j = j + 1
-- end
```

If the result type has been defined, the concatenated arrays have to be type compatible with the result type. If the result type is not known, the result type is defined by the usual assignments rules. Two arrays are said to be type compatible if either (1) the type of an array elements is a superclass of the type of the other array elements or (2) there exists a *well-known conversion* between the two arrays elements' types.

For consistency, the concatenation returns a first-level copy of the arrays. That is, the arrays' elements are not duplicated.

This proposal enhances code readability. By using the concatenation operator, we make our intent clear.

Although the concatenation of `char` arrays induces conversion to and from strings, it nonetheless remains possible and transparent for the unaware programmer; that is, if *a1*, *a2* and *a3* are three `char` arrays, `a3 = a1 || a2` is a valid construct and produces the expected result.

7.2. Deconstruction

NetRexx allows us to access the elements of an array individually but, to do so, we have to specify the element's index explicitly. It is not a real problem when we are accessing the first elements of the array, but it quickly leads to unreadable code when trying to access the last elements:

```
first = anArray[0]
second = anArray[1]
penultimate = anArray[anArray.length-2]
intimate = anArray[anArray.length-1]
```

This explicit specification of the element's index also implies that we have to take into account the initial index---hence the `-1` and `-2`---which is nonintuitive and can cause confusion. So, both this explicit specification and the need to repeat the operation for each and every elements we access obfuscate our intent and force us to write inelegant and repetitive code.

I suggest extending the **parse** instruction to allow the deconstruction of arrays by specifying a pattern that starts and ends with "`[`" and "`]`", respectively. In such cases, the term is expected to be an array. The pattern mimics an array initializer. Each element of the pattern is either a pattern or an extended "`...`" placeholder.

parse *term* '**[**' *pattern* [, *pattern*]... '**]**' **;**

There can be at most *one* extended placeholder in a given pattern level. An extended placeholder stands for as many elements as necessary (possibly negative).

examples:

```
foo = [1, 2, 3]
parse foo [a, b, c]                   -- a = 1, b = 2, c = 3
parse foo [..., last]                 -- last = 3
parse foo [a, b, ..., y, z]           -- a = 1, b = 2, y = 2, z = 3
parse aStringArray [hh1 '/' mm1 '/' ss1, hh2 '/' mm2 '/' ss2, ...]
parse [x, y] [y, x]
parse foo [..., elem, ...]            -- error!  only one "..." allowed
```

This notation is simple and intuitive, yet powerful and nonambiguous. It adds no incompatibility with existing code, and is in many ways similar to the notation used for string deconstruction---hence it does not unnecessarily complicate the language.

The types of the various elements can be checked at compile time. If the array to deconstruct contains too many (or too few) elements, the system's behavior may either mimic the one of string deconstruction or raise exceptions. Both behaviors are interesting. Comments welcome!

If concatenation of arrays is added (see Section 7.1), this specification can be extended to support it, so that we could write code such as `parse anArray [a, b] || tail`.

It may also be interesting to add positional patterns to array deconstruction, but, although quite powerful, this feature can be difficult to master. Then, it is already the case of positional patterns. So, why not?

7.3. Loops

This section contains suggestions for enhancing loops. Although mostly about array support, the predicates and **in**-loops sections are of more general interest.

7.3.1. Predicates

It is not uncommon to have to enclose the complete body of a loop within an **if** instruction when only some elements are of interest, as shown in Example 7.1. Alas, the intent of such a construction is not clear: If the body of the loop is somewhat long, we do not know at once whether there is an **else** clause for the **if** instruction, or if there are instructions following the **if** block.

Example 7.1:

```
loop i = 0 for content.length
  if content[i].hasChanged then do
    Transcript.log('Saving' content[i])
    content[i].save
  end
  else
    iterate                           -- not really needed
end
```

So I propose the extension of the predicate construct introduced in Section 2.3 to loops. This extension is made by allowing an optional predicate to appear in any loop instruction, before the optional *conditional* phrase. It is purely equivalent to enclosing the body of the loop in an `if predicate then ...` structure.

loop [**label** *name*] [**protect** *termp*] [*repetitor*] [*predicate*] [*conditional*] **;**

where *predicate* is:

'**|**' *expression*

Example 7.2 is the equivalent of Example 7.1, expressed with a predicate. The intent is clear (and the code more readable). We immediately know that only some elements are of interest.

Example 7.2:

```
loop i = 0 for content.length | content[i].hasChanged
  Transcript.log('Saving' content[i])
  content[i].save
end
```

The addition of a predicate phrase to the loop instruction does not break existing valid code. It makes the code more readable without significantly complexifying the loop instruction.

Again, "|" may not be the most effective notation to use for predicates. I am starting to find **when** more to my liking. If **when** is used, the predicate can follow the optional conditional phrase.

7.3.2. **over** loops

NetRexx offers **over** loops to walk through the sub-values of indexed strings. Arrays share many similarities with indexed strings, but cannot be used in **over** loops. This is inconsistent and unnecessarily forces us to deal with the arrays implementation's details.

I suggest extending the **over** loop to support arrays. This way, we could write code such as:

```
anArray = [10, 30, 60, 20, 40, 50]      -- currently,
loop i over anArray                     --  loop i = 0 for anArray.length
  say anArray[i]                        -- or
end                                     --  loop i = 0 to anArray.length - 1
```

This makes the code easier to read (the intent is simple to understand) and prevents us from possibly omitting the -1 correction factor. It also addresses the initial index possible error---that is, is this initial index 0 or 1?

The order in which the indices are enumerated is not specified. In particular, there is *no* guaranty two implementations will use the same order. The only guaranteed behavior is that all indices are enumerated (once and only once).

This proposal does not break existing code and improves consistency. It also reduces the perceived difference between arrays and indexed strings.

7.3.3. **in** loops

Sometimes when stepping through collections and or structures, we are only interested by the values (rather than their indices). For example, if we want to find the extremes in a numeric array, we have to write something like:

Example 7.3:

```
max = anArray[0]
min = anArray[0]
loop i = 0 for anArray.length           -- or loop i over anArray
  if max < anArray[i] then
    max = anArray[i]
  else
  if min > anArray[i] then
    max = anArray[i]
end
```

We have to deal with the index, which is strictly speaking unnecessary and can lead to confusion and errors. So I suggest adding the following to the *repetitor* phrase in loops:

> *vari* **in** *expri*
>
> where *vari* is a nonnumeric *symbol*
>
> and *expri* is an *expression* resulting in either an indexed string or an array

where *vari* successively takes all the available values in *termi*. The type of *vari* would be the type of the elements contained in *termi* (the array element type for arrays, etc.).

The order in which the elements are enumerated is not specified. In particular, there is *no* guaranty two implementations will use the same order. The only guaranteed behavior is that all elements are enumerated (once and only once).

By using this new construct, we can rewrite Example 7.3 as follow:

Example 7.4:

```
max = anArray[0]
min = anArray[0]
loop elem in anArray
  if max < elem then
    max = elem
  else
  if min > elem then
    min = elem
end
```

Our intent in Example 7.4 is clearer. The code is simpler and possibly more efficient, in that accessing an array element can be a costly operation.

This suggestion adds a new construct to NetRexx, but should not break existing code.

An implementation may decide to extend this construction to handle collections from the libraries, such as `Dictionary`s, `Vector`s, and so on.

The upcoming revision of Java (Java 2 v1.5, aka. Tiger) is likely to include a very similar mechanism (cf. http://jcp.org/aboutJava/communityprocess/jsr/tiger/enhanced-for.html).

We need to address the assignability of *vari*.

I am somewhat tempted to add a new notion, *segment*, which is of the form *exprn* `..`[*exprm*] [**by** *inc*], so that one can write code such as `loop i in 1..[10] [by 1]`, but this brings nothing---except readability and that segments could be objects---compared to `loop i = 1 [to 10] [by 1]`.

7.4. Iterators

By the mean of *array initializers*, we can create simple arrays. Alas, this construct requires us to enumerate all the expressions constituting the array elements. This limitation implies that we have to know the number of elements in the array at coding time and that the array does not contain many elements---otherwise, the code quickly becomes unreadable.

Hence, to complete the support of arrays as first-class citizens, I suggest introducing *iterators*, which are a quick and efficient way to create arrays.

'**[**' *expr* *repetitor* ['**,**' *repetitor*]... '**]**'

where *repetitor* is one of:

for *exprr* [*conditional*] [*predicate*]
for *varc* = *expri* [**to** *exprt*] [**by** *exprb*] [**for** *exprf*] [*conditional*] [*predicate*]
for *varo* **over** *termo* [*conditional*] [*predicate*]
for *vari* **in** *termi* [*conditional*] [*predicate*]
while *exprw*
until *expru*

and *conditional* is either of:

while *exprw*
until *expru*

and *expr*, *exprr*, *expri*, *exprt*, *exprb*, *exprf*, *exprw* and *expru* are *expressions*.

This new construct, a form of array initializer, consists of a *collation expression*, *expr*, followed by one or more *repetitors*. The value returned by the collection is an array of elements, one for each iteration of the collation expression.

The successive *repetitors* are equivalent to nested **loop**s. For example, for the following iterator:

```
  foo = [i+j+k for i = 1 to 5, for j = 1 to 5, for k = 1 to 5]
```

NetRexx may execute the following:

```
  foo = Rexx[125]
  $idx = 0
  loop i = 1 to 5
    loop j = 1 to 5
      loop k = 1 to 5
        foo[$idx] = i+j+k
        $idx = $idx + 1
      end
    end
  end
```

The size of the defined array has to be bounded. It cannot be infinite, but it may only be determinable at runtime. If an implementation detect an infinite iterator at compile time, an error is issued at compile time.

If **for** or **while** is a defined variable name, then the presence of this word is not recognized as forming the beginning of a *repetitor* phrase. (This behavior is consistent with current practices and preserves backward compatibility.)

Here follow some examples of iterators:

```
  changedAreas = [a for a in content | a.hasChanged]
  oneToTen = [i for i = 1 to 10]
    -- i.e. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  matrix = [['x**'i '+' j for i = 1 to 3] for j = 10 to 12]
    -- i.e. [['x**1 + 10', 'x**2 + 10', 'x**3 + 10'],
    --       ['x**1 + 11', 'x**2 + 11', 'x**3 + 11'],
    --       ['x**1 + 12', 'x**2 + 12', 'x**3 + 12']]
  [i**j for i = 1 to 5, for j = 2 to 3]
    -- i.e. [1, 1, 4, 8, 9, 27, 16, 64, 25, 125]
    -- aka  [1**2, 1**3, 2**2, 2**3, 3**2, 3**3, 4**2, 4**3, 5**2, 5**3]
```

Iterators are surprisingly intuitive and efficient tools. By using them, we can write very concise and clear code. Iterators allow us to think in highly descriptive ways. We no longer have to think or worry about the arrays clumsy implementation details---defining their sizes, taking care of their bounds, and so on. To end this section, I cannot resist the pleasure of giving the example of a quicksort procedure coded with iterators:

```
  method qsort(int[] array | array.length = 0) returns int[]
    return array

  method qsort(int[] [head] || tail) returns int[]
    return qsort([x for x in tail | x < head]) -
        || [head] -
        || qsort([x for x in tail | x >= head])
```

If the compiler finds that the actual construction of the array is unnecessary, it can transform the code in an equivalent construct. This hint especially applies to local arrays. It naturally also concerns the currently supplied literal arrays. The equivalent---but more efficient---implementation can be done via local variables or added loops, for example.

7.5. Summary

By making arrays "real" first-class citizens in NetRexx, we would immensely enhance the expressivity and descriptiveness of the language. This enhanced expressivity allows us to think at a higher level. It also reduces the clumsiness of our code.

The first five proposals (namely, concatenation, predicates, **over** loops and **in** loops) are somewhat *classic*, in that they mostly are enhancements of existing constructs. The last proposal (the iterators) is different. Very few existing languages offer something similar (if we exclude LISP and its declinations). I do not think it would be judicious to add such a feature to, say, Java. The concept would just not fit. But NetRexx is much more descriptive than Java. The gap to fill is smaller. So I think it is indeed a valuable addition for NetRexx.

There is another feature that could be added. The possibility of addressing a subset of an array. For example, something like `foo[1..5]` would return an array of five elements. Such subsets would either return a value (an array) or represent a *restricted view* of an array (and, as such, could be used in the left-hand side of an assignment). Specifying a segment (see Section 7.3.3) or an array in the brackets would be a simple way to denote such subsets.

http://java.sun.com/aboutJava/communityprocess/jsr/jsr_083_multiarray.html

(sections) JSR-000083 JavaTM Multiarray Package

8. Default implementations in interfaces

It is not uncommon, when creating applications, to have many classes sharing a given interface. Sometimes, we would also like to share the implementation of one or more method(s) but, due to NetRexx's single inheritance, doing so is not always possible. Similarly, when designing interfaces, we sometimes feel the need to provide a *default implementation* for a method.

I suggest allowing an optional default implementation for functions defined in interfaces. Those default implementations, when present, would be used in place of the ones generated by **adapter** classes (but not only in **adapter** classes).

A default implementation can only call methods defined in the interface (or in one of the interfaces it implements and or extends, if any).

Default implementations are safe in that they are backward compatible with existing code, and in that they do not change the semantic of interfaces. Unlike noninterface classes, interfaces cannot maintain a state on their own. It means that default implementations in interfaces differ significantly from multiple inheritance, as they cannot alter the state of the object they are members of by themselves.

Albeit sharing a superficial resemblance with abstract classes, default implementations in interfaces are conceptually different. They do not serve the same purpose. An abstract classe is a base class upon which we (eventually) build concrete classes. When a class implements an interface, it means this class has a given characteristic. An interface is not a base class upon which we build concrete classes. It is a way to denote that some classes have and share a given characteristic.

By using default implementations in interfaces, we enhance our productivity. We do not have to repeatedly write the same code. The expressivity of NetRexx is increased. We can write more extensive interfaces, with more generic methods, without having to worry about implementing those generic methods again and again.

This proposal requires a native NetRexx compiler in that it relies on features not present in the Java language, but permitted by the Java Class file format. A possible implementation is by the mean of attributes in the class file.

9. Delegates

The delegation pattern is of common use in the Object Oriented Programming world. Although nothing in NetRexx prevents us from using delegation, all the work is left in the hands of the programmer---namely ours.

Let us take the case of a bean. If we create a bean with bound properties, the JavaBean specification requires us to support `PropertyChangeListeners` and to fire `PropertyChangeEvents` each time a bound property is modified.

This is a repetitive task and, to prevent as much errors as possible, the JavaBean library contains a `PropertyChangeSupport` class that provides a default implementation to handle the registration of

`PropertyChangeListeners` and the `PropertyChangeEvents` dispatching.

We can either subclass the `PropertyChangeSupport` or use an instance of this class to delegate the work to it. But subclassing is not always possible, as Java and NetRexx have no support for multiple inheritance (if the bean is, say, a `Container`, it cannot be a `PropertyChangeSupport` at the same time).

So we usually have to add the following piece of code to the bean:

Example 9.1:

```
properties private
pcs = PropertyChangeSupport(this)

method addPropertyChangeListener(pcl = PropertyChangeListener) protect
  pcs.addPropertyChangeListener(pcl)

method removePropertyChangeListener(pcl = PropertyChangeListener) protect
  pcs.removePropertyChangeListener(pcl)

method firePropertyChangeEvent(name = String, -
                               oldValue = Object, -
                               newValue = Object)
  pcs.firePropertyChangeEvent(name, oldValue, newValue)
```

This coding is quite long and repetitive, especially if we define many beans. Worse, if a new method is added to the `PropertyChangeSupport` class, we have to update the code of each and every bean in which we implement such a delegate.

The design I present here for support of delegation tries to remove as much unnecessary work as possible for the programmer, while preserving consistency and predictability. Using delegates, we can rewrite Example 9.1 as

```
properties delegate
pcs = PropertyChangeSupport(this)
```

and, if the definition of `PropertyChangeSupport` changes, a simple recompilation will usually be sufficient.

The **properties** instruction [NRL 105] is used to define the attributes of following *property* variables. The *visibility* of properties may include a new alternative: **delegate**. Properties with this form of visibility are known as *delegates*. These are properties of a known type that are private to the class, but which are expected to provide some service.

The delegate attribute of a property is an alternative to the **public**, **private**, **indirect**, and **inheritable** attributes. Like private properties, delegates can only be accessed directly by name from within the class in which they are defined; other classes can only access them using the services they provide (or other methods that may use, or have a side-effect on, the properties).

Delegates may be **transient** or **static**. They may not be **constant** or **volatile**.

In detail, the rules used for generating automatic methods for a property whose declared type is `xxxx` are as follows:

1. A wrapper is generated for all public instance methods exposed by the `xxxx` class and its superclasses (up until a common superclass is found with the current class). If the property is **static**, these methods will be generated as **static** methods.
2. If a class defines more than one delegate and if two of those delegates define an identical method, an error is issued. To circumvent this, an explicit implementation has to be provided for all such ambiguous methods.

It is an error to declare a delegate for which no wrapper can be generated if the delegate is not used in the current class code.

A delegate may have a real type which is either a subtype of its declared type or an implementation of it. As previously stated, wrappers are only generated for methods defined in the declared type. For example, in

```
properties delegate
output = DataOutput RandomAccessFile("foo", "rw")
```

the generated methods are the ones defined by the `DataOutput` interface.

Note that in all cases a method will only be generated if it would not exactly match a method explicitly coded in the current class.

To complete this section, let us build a simplistic instance counter. To do that, we define an `InstanceCounter` class to handle the counting:

```
class InstanceCounter

  properties indirect constant
  instanceCount = 0

  method incInstanceCount protect
    instanceCount = instanceCount + 1
```

To add an `instanceCounter` to a class, we define a static delegate in the class. Then, each time a new instance is created, we call the `incInstanceCount` (now) static method. (In this example, the counter is primitive, as it also includes subclasses instances. It can easily be enhanced, but it is not the point we are trying to make here.)

```
class ATest extends Component

  properties delegate static
  cpt = InstanceCounter()

  method ATest
    incInstanceCount

  ...
```

We can then at any time query the number of created instances:

```
aTest1 = ATest()
...
say ATest.getInstanceCount
```

Delegates simplify our work. We no longer have to repeatedly write wrappers. The code we write is clear, concise, and usually immune to delegate changes (possibly modulo a recompilation). By using delegates, we operate at a higher level.

10. Summary

The eight areas of enhancements I addressed in this document share one purpose: to make NetRexx easier to use. Let us review their benefits one last time:

- Piece-wise methods definitions enhance the expressivity of NetRexx. The code we write is more descriptive, and our intent is more obvious.
- Privileged objects make our code more concise without impeding its readability. In fact, it possibly reduces the number of errors we may make if we inadvertently confuse two similarly-named variables.
- An enhanced type system enforces stricter type safety at compile time. It also enhances our productivity as we no longer have to type the variables or duplicate similar methods. We can also adapt base classes we otherwise have no control upo.
- Short-circuit logical operators reduce the complexity of our code. Our intent is simpler to understand, and no unnecessary code duplication occurs.
- Assertions are a powerful way to ensure programs' correctness. They also provide additional documentation for the code we write.
- "First-class" arrays immensely enhance the expressivity of NetRexx. They allow us to write much more descriptive code. We can think at a higher level.
- Default implementations in interfaces avoid code repetition. It allows us to write code more efficiently.
- Delegates implement the delegation pattern. This design pattern is common, and providing support for it, although somewhat ad-hoc, is useful. Delegates support allows us to write simpler and clearer code.

My ramblings come to an end. I hope you have found at least some of them not totally uninteresting. The ball is now in **your** hands. Please send in your comments and suggestions. They are essential. I am looking forward to read them.

Acknowledgments

I am grateful to Mike Cowlishaw for inventing NetRexx and making numerous suggestions and comments on earlier drafts of this document.

I thank Dion Gillard for his support, and Alan Ackerman and Myron Chaffee for their comments.

I thank all those who reviewed previous drafts of this document and were delicate enough not to send their comments to preserve my sensibility :-)

Bibliography

[HINDLEY]	Hindley, J.R. 1969. The principal type-scheme of an object in combinatory logic. *Trans. AMS* 146, 29--60.
[MILNER]	Milner, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17, 348--75.
[NRL]	Cowlishaw, M.F. 1997. *The NetRexx Language*. Prentice Hall, ISBN 0-13-806332-X.

History

0.101	More clarifications on iterators specification. Minor changes in the bounded types section. Typos corrected.
0.100	Macros replaced by type synonyms. Iterators specification made more consistent. Generics support proposed.
0.99	Reworked bounded type section, now using the & operator. Added references to the Tiger java proposal (aka Java 2 v1.5).
0.98	Added references in the Assertions section. Label clause now formally handled. A more interesting example. Minor fix in the Array deconstruction section. (The diagram was incorrect in that parse was followed by an expression instead of a term.) Minor update in the Short circuit logical operators section, to address the recent addition of a comma operator in if and when instructions. Removed a paragraph in the introduction, as an interpreter now exists for NetRexx.
0.96	Added an introduction Clarifications added in Piece-Wise method definitions section Object deconstruction subsection removed. Array deconstruction subsection moved to the Array section. Shortcuts section renamed Privileged Objects The Type inference algorithm section has been replaced with a Type System section. Two new proposals added (Bounded types and Views). Shortcuted logical operators section renamed Short-circuit logical operators. Clarifications added. Minor changes in the Assertions section. Loop variants and loop invariants discussion removed. Clarifications added. Clarifications added the Array section. It now includes the Array deconstruction subsection (formerly in the Object deconstruction section). Minor changes in the Macro section Clarifications in the Default Implementation in Interfaces section Delegate proposal text included Added a Summary and Acknowledgments section
0.94	Declared exceptions addressed in Piece-wise function definitions section Evaluations in Shortcut section addressed Minor layout change to more accurately support non--CSS-aware HTML user agent
0.92	Concatenation of arrays section renamed Arrays and modified Typos corrected Predicates added Explanations added in the Assertions section Implicit this keyword section revised and renamed Shortcuts Macros proposal modified
0.90	Initial public release

Martin Lafaix