Defaulting and type conversion

Unicon provides a syntactic structure to specify type conversions and default values. The general, per-parameter form is this:

parameter-name : *conversion-procedure* : *default-value*

Both *conversion-procedure* and *default-value* are optional.

Here's an example that uses only a conversion procedure:

```
class Rectangle(width, height)
    initially(w:integer, h:integer)
        width := w
        height := h
end
```

If the value supplied for `w` or `h` is not convertible to an integer, (i.e., if `integer(...)` fails) error 101 is produced:

```
][ r := Rectangle(3, "four");
Run-time error 101
integer expected or out of range
offending value: "four"

][ r := Rectangle();
Run-time error 101
integer expected or out of range
offending value: &null
```

Note that this specification can be used with both methods and ordinary procedures.

Question: What's the real benefit of this language element?

Defaulting and type conversion, continued

For reference:

parameter-name : *conversion-procedure* : *default-value*

Recall that `split()`'s second argument defaults to the character set containing a blank and a tab.

Instead of this:

```
procedure split(s, c)
    /c := ' \t'
    ...
```

We could do this:

```
procedure split(s, c:' \t')
    ...
```

We could further constrain the argument values by specifying conversion routines:

```
procedure split(s:string, c:cset:' \t')
    ...
```

Note that only a literal is permitted for the default value.

Problem: What's wrong with the following routine?

```
procedure f(x:list)
    ...
```

Defaulting and type conversion, continued

A user defined procedure may be specified as the conversion routine.

If the routine fails, then a run-time error is produced. If it succeeds, the value returned is passed as the argument value. (Just as with a built-in routine like `integer`.)

Example:

```
procedure f(n:odd)
    return n * 2
end

procedure odd(x)
    if x % 2 = 1 then return x
end
```

Usage:

```
][ f(5);
   r := 10  (integer)

][ f(20);
Run-time error 123
invalid type
offending value: 20
```

Problem: There's no way to do something like this:

```
procedure f(x:(integer|string))
    ...
```

How could that effect be achieved?

The `xcodes` facility

The `xcodes` package in the IPL allows a nearly arbitrary data structure to be written to a file and later restored.

Here is a program that generates a random list and saves it to a file using `xencode()`:

```
link xcodes, random
procedure main()
    randomize()

    L := randlist(10, 15)
    write("List: ", ltos(L))

    f := open("randlist.out", "w")

    xencode(L, f)

    close(f)
end
```

Execution:

```
% xcodes1w
List: [29,97,[34,92],[[63,6]],63,35,13]

% cat randlist.out
L
N7
N29
N97
L
N2
N34
N92
L
N1
...a few lines more...
```

The `xcodes` facility, continued

Here is a program that loads <u>any</u> structure written with `xencode()`:

```
link xcodes, image
procedure main(args)
    f := open(args[1]) | stop("Can’t open file")
    S := xdecode(f)
    write("Restored structure: ", Image(S))
end
```

Execution:

```
% xcodes1r randlist.out
Restored structure: L1:[
  29,
  97,
  L2:[
    34,
    92],
  L3:[
    L4:[
      63,
      6]],
  63,
  35,
  13]
```

`xencode()` can't accurately save co-expressions, windows, and files, but allows them to be present in the structure.

Problem: How can a facility like `xencode`/`xdecode` be written?