Unicon—History

One predecessor of Unicon is Idol, Icon-derived Object Language.

Idol was developed at the University of Arizona by Clint Jeffery in 1988 for a graduate course on object-oriented programming.

"Unicon" initially stood for "UNIX Icon"—a version of Icon with a set of POSIX extensions by Shamim Mohamed developed in 1997. Mohamed learned Icon at the U of A but, because of Icon's lack of access to many OS facilities, used Perl for a variety of systems programming tasks. He wrote:

"While it is true that Perl substitutes for a conglomeration of sed, awk and shell scripts, it does so with some of the worst language features from them."

Unicon was his solution.

In 1999 Jeffery and Mohammed merged their work and other elements, such as an ODBC interface, into a single system, which was tentatively called Icon-2.

The name Unicon was later recycled, now standing for "Unified Extended Icon".
Class and method basics

Here is a simple Unicon class that models a coordinate-less rectangle:

```unicon
class Rectangle(width, height)
    method area()
        return width * height
    end

    method perimeter()
        return width*2 + height*2
    end

    method str()
        return "Rectangle(" || width || "x" || height || ")"
    end
end
```

The class name is `Rectangle`.

It has two *attributes* (or *fields*), `width` and `height`.

It has three methods: `area`, `perimeter`, and `str`.

The `str` method produces a value such as
"Rectangle(3x4)"
Class and method basics, continued

For reference:

```
class Rectangle(width, height)
  method area()
    return width * height
  end
end
```

We can create instances of `Rectangle` like this:

```
r := Rectangle(3,4)
Rs := [Rectangle(3,4), Rectangle(5.0,7)]
r2 := Rectangle("3.4", 7)
```

For this class the constructor is essentially a record constructor—the supplied values are assigned directly to the fields `width` and `height`.

Methods are invoked with a familiar syntax:

```
a := r.area()
write("Perim: ", r.perimeter())
every write((!Rs).str())
write(Rectangle(2.9, 9.02).perimeter())
```
Class and method basics, continued

Just like any other Icon procedure call or record construction, no error checking is done. A null value is used for missing arguments and extra arguments are ignored.

All of the following execute without error:

\[
\begin{align*}
  r1 & := \text{Rectangle}(3); \\
  r2 & := \text{Rectangle}(\text{"abc"}, \text{"xyz"}); \\
  r3 & := \text{Rectangle}(7, 9, \text{"abc"});
\end{align*}
\]

Question: Which methods work for which of the above instances?

Unicon has no provision for access specifications like "public" and "private"—all attributes and methods are accessible in any context. This works:

\[
\begin{align*}
\text{procedure main()}
  \quad rr & := \text{Rectangle}(3,4) \\
  \quad rr.\text{width} & := 20 \\
  \quad rr.\text{height} & := 30 \\
  \quad \text{write}(rr.\text{area}()) \\
\end{align*}
\]

Question: How can encapsulation be enforced?
Class and method basics, continued

The constructor is a procedure and can be treated like any other procedure:

\[
R := \text{Rectangle} \\
r1 := R(5,7) \\
r2 := [R][1](3,4) \\
r3 := ("Rect" || "angle")(3,4);
\]
Class and method basics, continued

Here is a program that produces a memory fault on SunOS 5.9:

```unicon
class X()
   method f()
      write("in f()...")
   end
end

procedure main()
   x := X()
   x.f()
   x.g()
end
```

Execution:

```plaintext
% bogus
in f()...

Run-time error 302
File bogus.icn; Line 10
memory violation
Traceback:
   main()
   {record X__state_1(record X__state_1(2),
      record X__methods_1(1)) . g} from line 10
   in bogus.icn
```
The initially section

The simplistic behavior of assigning values in a constructor call to the attribute in the corresponding position is often inadequate.

An initially section can be added to trigger processing when the constructor is called.

```plaintext
class Rectangle(width, height, _area)
  method area()
    return _area
  end
  ...other methods...
  initially(w, h)
    write("initially: ",
          Image([width, height, _area],3))
    width := w
    height := h
    _area := w * h
  end
```

If present, initially must follow all methods.

The end that ends the class definition also ends the initially section.

```plaintext
][ rr := Rectangle(3,4);
 initially: L1:[&null,&null,&null]
     r := ...lots...

 ][ rr.area();
     r := 12  (integer)
```

If initially(...) is present, no attributes are automatically initialized.
Initially, continued

The initially section can be used to enforce constraints on the constructor's arguments.

```plaintext
class Rectangle(width, height, _area)
    ...
    initially(w, h)
        if /w | /h then fail
        if not numeric(w) | not numeric(h) then fail
        width := w
        height := h
        _area := width * height
    end
```

Execution:

```plaintext
][ rr := Rectangle(3);
    Failure

][ rr := Rectangle(3, "x");
    Failure

][ rr := Rectangle(3, "3.4");
    r := ...lots...
```

Note that by default an initially section succeeds.

Problem: There is no overloading of method names or the initially section. How could, for example, an omitted height default to the same value as the width?

```plaintext
r := Rectangle(3)
```
initially, continued

If there is a parameterless initially section then the arguments of the constructor call are used to initialize the attributes.

Example:

```plaintext
class Counter(count)
   method inc()
      count +:= 1
      return count
   end

   method value()
      return count
   end

   initially
      /count := 0
   end

Usage:

```plaintext
][ A := Counter(10);
   r := ...lots...

][ B := Counter();
   r := ...lots...

][ A.value();
   r := 10  (integer)

][ B.value();
   r := 0  (integer)
```
The implicit variable \texttt{self}

Unicon's counterpart for Java's \texttt{this} is \texttt{self}.

One use is to distinguish between attributes and parameters:

\begin{verbatim}
class Rectangle(_area, width, height)
    initially(width, height)
        self.width := width
        self.height := height
    ...
end
\end{verbatim}
Class specification—general form

Here is the general form of a class specification:

```plaintext
class classname(attribute1, attribute2, ..., attributeN)
    method method1(param1, param2, ..., paramN)
        ...code for method...
    end

    ...additional methods...

initially(param1, param2, ..., paramN)
    ...code to execute upon construction...
end
```

Note that all attributes are specified in the list following the class name.

Here is a minimal class definition:

```plaintext
class X()
end
```
Method result sequences

Methods may fail, or produce a single result, or be generative, just like regular Icon procedures. Imagine a \texttt{side()} method that generates the width and height of a rectangle:

```icon
class Rectangle(width, height, _area)
  ... 
  method side()
    suspend width | height 
  end  
  ...
end
```

Usage:

```icon
procedure main()
  rects := []
  every 1 to 20 do
    put(rects, Rectangle(?20, ?20))
  end 
  every r := !rects do
    if r.side() > 10 then
      write(r.str())
  end
end
```

Output:

```
Rectangle(7x11)
Rectangle(2x15)
Rectangle(2x15)
Rectangle(11x13)
Rectangle(12x15)
Rectangle(15x5)
...```
Circle drag/drop in Unicon

Recall this program from Graphics slide 31: (drag1)

```
record circle(x, y, r)
procedure main()
   WOpen("size=600,300","drawop=reverse")
   DrawLine(300,0,300,300)
   circles := make_circles()
   repeat case Event() of {
      &lpress:
         if c := point_in(circles, &x, &y) then {
            lastx := c.x; lasty := c.y
            r := c.r
            repeat case Event() of {
               &ldrag: {
                  DrawCircle(lastx, lasty, r)
                  DrawCircle(lastx := &x, lasty := &y, r)
               }
               &lrelease: {
                  DrawCircle(lastx, lasty, r)
                  if &x <= 300 then {
                     DrawCircle(&x, &y, r)
                     c.x := &x; c.y := &y
                  }
                  else
                     delete(circles, c)
                     break
               }
            }
         }
      }
   }
end
```
Circle drag/drop in Unicon, continued

Here is a version in Unicon. First, a Circle class:

```unicon
class Circle(x, y, r)
  method has_pt(pt_x, pt_y)
    if sqrt((x-pt_x)^2+(y-pt_y)^2) < r then
      return self
  end

  method move_to(new_x, new_y)
    erase()
    x := new_x; y := new_y
    draw()
  end

  method erase()
    draw()
  end

  method draw()
    DrawCircle(x, y, r)
  end

initially
  draw()
end
```

Note that the initially section counts on direct assignment of attributes from the constructor call.

The code above does not track the on-screen state (drawn or not) and thus places an additional responsibility on the caller.
Circle drag/drop in Unicon, continued

Main program:

```unicon
procedure main()
    WOpen("size=600,300","drawop=reverse")
    DrawLine(300,0,300,300)

    circles := make_circles()

    repeat case Event() of {
        &lpress:
            if c := (!circles).has_pt(&x, &y) then {
                repeat case Event() of {

                    &ldrag: c.move_to(&x, &y)

                    &lrelease: {
                        if &x <= 300 then
                            c.move_to(&x, &y)
                        else {
                            c.erase()
                            delete(circles, c)
                        }
                        break
                    }
                }
            }
    }

Which version is better?
Inheritance

Here is a simple general form for specifying inheritance:

```
class class-name : superclass-name (class-attributes)
 ...
end
```

Here is a skeletal three class hierarchy to model geometric shapes:

```
class Shape(name)
end

class Rectangle: Shape (width, height)
end

class Circle: Shape (radius)
end
```

Rectangle is a subclass of Shape and has three attributes: name, width, and height.

Circle is a subclass of Shape and has two attributes: name and radius.

In Unicon there is no common superclass such as Java's Object class.
Superclass initialization

If a subclass has no `initially` section then the superclass's `initially` section is called.

The superclass's `initially` section is NOT CALLED if the subclass has an `initially` section.

Example:

```plaintext
class Shape(name)
    initially
    write("Shape's initially")
end
class Circle: Shape (radius)
end
class Rectangle: Shape (width, height)
    initially
    write("Rectangle's initially")
end

procedure main()
    c := Circle(5)
    r := Rectangle(3,4)
end
```

Output:

```
Shape's initially
Rectangle's initially
```

If a subclass requires an `initially` section then it should explicitly invoke the superclass `initially` section.
Superclass initialization, continued

Here is an example of invoking a superclass initially section:

class Shape(name)
    initially(nm)
        name := \nm | "<none>
        write("Shape initially(), name = ", name)
    end

class Rectangle: Shape (width, height)
    initially(w, h, nm)
        write("Rectangle initially()")
        width := w
        height := h
        self$Shape.initially(nm)
    end

procedure main()
    r := Rectangle(3, 4)
    write(Image([r.name, r.width, r.height],3))

    r2 := Rectangle(5, 7, "B")
    write(Image([r2.name, r2.width, r2.height],3))
end

Output:

Rectangle initially()
Shape initially(), name = <none>
L1:['<none>',3,4]

Rectangle initially()
Shape initially(), name = B
L2:['B',5,7]

Note that there is no rule that specifies when superclass initialization must be done.
Method inheritance and overriding

Unicon's rule for method inheritance is a common one: Subclasses inherit superclass methods unless they supply their own version of a method.

class Shape()
    method area()
        end
end

class Rectangle: Shape (_width, _height)
    method area()
        return _width * _height
    end
end

class Circle: Shape (_radius)
end

procedure main()
    r := Rectangle(3, 4)
    c := Circle(5)

    write("r's area = ", r.area())
    write("c's area = ", c.area())
end

Output:

    r's area = 12
Abstract classes

Unicon provides no means to declare a class or method as abstract.

One way to ensure that a subclass overrides a method is to add code that produces an error if an overriding method is forgotten:

```unicon
class Shape()
    method area()
        stop("Shape.area() called!?")
    end
end
```

Question: Icon's association of type with values rather than variables implies that some errors are not detectable until the code is executed. Would it be possible to enforce an abstract declaration at compile time?
Inheritance and dynamic typing

Languages like Java use inheritance to allow code to be written in terms of a superclass and then be run with subclass instances.

```java
public static Shape biggestArea(Shape shapes[]) {
    if (shapes.length == 0) return null;
    Shape it = shapes[0];
    for (int i = 1; i < shapes.length; i = i + 1) {
        if (shapes[i].getArea() > it.getArea())
            it = shapes[i];
    }
    return it;
}
```

Because of Icon's value-based typing, inheritance is not needed to write such code.

In the following code there is no common superclass for A and B, but the routine show_wha(t)() can a handle a list of As, Bs, and any other objects that have a what() method.

```icon
class A()
    method what()
        return "I'm an A!"
end
end

class B()
    method what()
        return "I'm a B..."
end
end

procedure show_wha(t)(L)
    every o := !L do
        write(o.what())
end
```
Multiple inheritance

Unicon supports *multiple inheritance*—a class can have any number of superclasses. Here's an abstract example:

```unicon
class A(_a)
  method f()
    write("A.f()")
  end
end

class B(_b1, _b2)
  method g()
    write("B.g()")
  end
end

class C(_c)
end

class D(_d1, _d2, _d3)
  method h()
    write("D.h()")
  end
end

class ABC: A : B : C (_abc1)
  method g()
    write("ABC.g()")
  end
end

class M : D : ABC (_m1, _m2)
end
```

A subclass inherits all attributes and methods of all its superclasses.

```unicon
procedure main()
  abc := ABC()
  abc.f()  # calls A.f()
  abc.g()  # calls ABC.g()

  m := M()
  m.f()   # calls A.f()
  m.g()   # calls ABC.g()
  m.h()   # calls D.h()
end
```
Multiple inheritance, continued

A less abstract example—a **DrawableRectangle**:

```unicon
class Drawable(_x, _y)
    method draw()
        stop("Drawable.draw() not overridden")
    end

    initially(x,y)
        _x := x; _y := y
    end

class DrawableRectangle : Rectangle : Drawable()
    method draw()
        DrawRectangle(_x, _y, _width, _height)
    end

    initially(w, h, x, y, nm)
        self$Rectangle.initially(w, h, nm)
        self$Drawable.initially(x, y)
    end

tool procedure main()
    WOpen("size=300,300")
    rects := []
    every i := 1 to 20 do
        put(rects, DrawableRectangle(?40, ?40, ?300, ?300))
    end

    every r := !rects do
        if r.area() < 1000 then
            r.draw()

    WDone()
end
```

CSc 451, Spring 2003
W. H. Mitchell
Class variables and methods

Unicon does not have support for class variables and methods.

Problem: What is the essence of class variables and methods and how can they be approximated/simulated?
Class variables and methods, continued

Here is a version of the Rectangle class that uses a global variable to loosely simulate a class method that returns the number of rectangles that have been created.

```unicon
class Rectangle(width, height)
  method area()
    return width*height
  end

initially
  initial{
    Rectangle_num_created := 0
  }
  Rectangle_num_created +:= 1
end

global Rectangle_num_created

procedure Rectangle_created()
  return Rectangle_num_created
end

procedure main()
  every 1 to 20 do
    Rectangle(?100, ?100)

    write(Rectangle_created(),
      " rectangles created")
  end

What are the pros and cons of this approach?
```
Class variables and methods, continued

Another approach is to use a method with a static variable and have a parameter serve as a flag indicating whether the value should be fetched or modified.

```unicon
class Rectangle(width, height)
...
method created(increment)
    static created
    initial created := 0
    if \increment then
        created +:= 1
    else
        return created
end

initially
    created(1)    # any non-null value would do
end

procedure main()
    every 1 to 20 do
        Rectangle(?100, ?100)
        write(Rectangle().created(),
        " rectangles created")
end
```

What are the pros and cons of this approach?
Class variables and methods, continued

Here is another approach:

```plaintext
class Rectangle(width, height)
  initially
  initial {
    if type(Rectangle_class) == "procedure" then
      Rectangle_class()
    }
    Rectangle_class.new_instance()
  end

class Rectangle_class(num_rects)
  method created()
    return num_rects
  end
  method new_instance()
    num_rects += 1
  end
  initially
    Rectangle_class := self
    num_rects := 0
end

procedure main()

  every 1 to 20 do
    Rectangle(?100, ?100)

    write(Rectangle_class.created(),
      " rectangles created")
  end

What are the pros and cons of this approach?
```

What are the pros and cons of this approach?
Behind the scenes in Unicon

Unicon programs are preprocessed, yielding a syntactically valid Icon program that is then compiled with `icont`. The resulting bytecode executable can then be run on the Unicon virtual machine.

A Unicon method is translated into an Icon procedure that has the class name prepended and an initial argument of `self`.

The methods in this Unicon class:

```unicon
class Rectangle(width, height)
    method area()
        return width * height
    end
    method set_width(w)
        width := w
    end
end
```

are translated into this Icon code:

```icon
procedure Rectangle_area(self)
    return self.width * self.height
end

procedure Rectangle_set_width(self, w)
    self.width := w
end
```

CSc 451, Spring 2003
W. H. Mitchell
Behind the scenes in Unicon, continued

Here is the balance of the generated Icon code for the class:

```icon
record Rectangle__state(__s, __m, width, height)
record Rectangle__methods(area, set_width)
global Rectangle__oprec

procedure Rectangle(width, height)
  local self, clone
  initial {
    if /Rectangle__oprec then
      Rectangleinitialize()
  }
  self := Rectangle__state(&null, Rectangle__oprec, width, height)
  self.__s := self
  return self
end

procedure Rectangleinitialize()
  initial Rectangle__oprec :=
    Rectangle__methods(Rectangle_area,
                        Rectangle_set_width)
end

For r := Rectangle(3, 4) here is the picture:
```

![Diagram of the generated code for the Rectangle class](image)
Behind the scenes in Unicon, continued

For reference:

record Rectangle__state(__s, __m, width, height)
record Rectangle__methods(area, set_width)

Here is a main program. The Unicon preprocessor makes no changes in it:

procedure main()
    r := Rectangle(3, 4)
    r.set_width(7)
    write("Area: ", r.area())
end

Recall that the type of r is Rectangle__state and note that there is no area field in that record.

What happens is this: When the field operator (binary period) detects that r has no field named area, it looks to see if the first field of r is named __s. If so, it then looks in the record referenced by the second field (__m) for a field named area and if found, the value of the field is the result of evaluating r.area.

To see the result of Unicon preprocessing, use the -E flag:

    unicon -E myclass.icn
Access to system services

The object-oriented programming facilities are one aspect of Unicon. Another is Unicon's access to operating system services.

One of the services available is the **stat()** system call, which produces a variety of information about a file. Unicon's **stat(fname)** call returns a record with the following information (and more) about the file `fname`:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>ID of device containing the file</td>
</tr>
<tr>
<td>ino</td>
<td>Inode number</td>
</tr>
<tr>
<td>mode</td>
<td>File mode (e.g. protections)</td>
</tr>
<tr>
<td>nlink</td>
<td>Number of links</td>
</tr>
<tr>
<td>uid, gid</td>
<td>User-id and group-id</td>
</tr>
<tr>
<td>size</td>
<td>Size of the file in bytes</td>
</tr>
<tr>
<td>atime</td>
<td>Time of last access</td>
</tr>
<tr>
<td>mtime</td>
<td>Time of last modification</td>
</tr>
<tr>
<td>ctime</td>
<td>Time of last inode change</td>
</tr>
<tr>
<td>symlink</td>
<td>If a symbolic link, the name linked to.</td>
</tr>
</tbody>
</table>
Example: List files by size

`bysize` is a program that uses `stat(fname)` to produce a list of files in a named directory sorted by file size in descending order:

```
% bysize /home/cs451/a5
10663 mtimes
  3730 day
  3461 mtimes.1
  3450 mcycle
  701 mcycle.2
  632 mtimes.2
  562 mtimes.ex
  229 tmtimes.sh
  148 mcycle.1
  104 mtimes.3
```

An `ls`, for comparison:

```
% ls -la /home/cs451/a5
total 60
drwxr-sr-x 3 whm cs451 4096 Apr 21 03:15 .
drwxr-sr-x 19 whm cs451 4096 Apr 16 03:40 ..
drwx------ 2 whm cs451 4096 Feb 12 22:33 v1
-r-xr-xr-x 1 whm cs451 3461 Feb 12 04:41 mcycle
-r-xr-xr-x 1 whm cs451 3450 Feb 12 21:54 mcycle.1
-r-xr-xr-x 1 whm cs451 701 Feb 12 21:54 mcycle.2
-r-xr-xr-x 1 whm cs451 10663 Feb 12 04:42 mtimes
-r-xr-xr-x 1 whm cs451 3461 Feb 12 04:41 mtimes.1
-r-xr-xr-x 1 whm cs451 632 Feb 12 04:41 mtimes.2
-r-xr-xr-x 1 whm cs451 104 Feb 12 04:41 mtimes.3
-r-xr-xr-x 1 whm cs451 562 Feb 12 04:41 mtimes.ex
-r-xr-xr-x 1 whm cs451 229 Feb 12 04:41 tmtimes.sh
```

Note that `bysize` does not show the three directories (., .., and v1)
record file_info(name, size)  # name and size of a file

procedure main(args)
#
# Change to the directory named on the command line
chdir(args[1]) |
      stop(args[1], ": Bad directory")
#
# A directory can be opened like a file. Reading from a directory
# produces the entries in the directory.
dir := open(".")

files := [ ]
#
# Read each directory entry and stat it. If an entry is not a directory,
# add it to the list.
#
while fname := read(dir) do {
   stat_rec := stat(fname)

   #
   # If not a directory, include it.
   #
   if stat_rec.mode[1] == "d" then
      put(files, file_info(fname, stat_rec.size))
   }

   #
   # Sort by file size and print.
   #
   files := sortf(files, 2)
   every r := files[*files to 1 by -1] do
      write(right(r.size,9)," ", r.name)
end
Example: A simple shell

An interesting application of Unicon's system service facilities is a simple command processor, commonly called a shell, that is used to invoke programs.

UNIX shells use a "fork and exec" sequence to start programs.

The call fork() creates a child process that is a copy of the current process. In the parent process, fork() returns the process id of the child. In the child process, fork() returns zero.

Example:

```
procedure main()
    if fork() = 0 then
        write("child process id is ", getpid())
    else
        write("parent process id is ", getpid())

    write("Hello, world!"

end
```

Output:

```
parent process id is 7713
Hello, world!
child process id is 7716
Hello, world!
```

Note that fork creates a process, not a thread—there's no sharing of memory between the two processes.
A simple shell, continued

Here is a larger example with `fork()`. Both the parent and child process identify themselves and then do three random sleeps (`delay()`)s, printing the time when they awake.

```link random
procedure main()
    if fork() = 0 then who := "child"
    else who := "parent"

    randomize()
    write(who, " process id is ", getpid())
    every 1 to 3 do {
        delay(?10000)
        write(who, " @ ", &clock)
    }

    write(who, " done")
end
```

Output:

```
% fork
parent process id is 8730
child process id is 8733
child @ 03:43:46
parent @ 03:43:49
parent @ 03:43:49
child @ 03:43:53
parent @ 03:43:57
parent done
% child @ 03:43:59
child done
```

Questions:

(1) Why is there a "%" in the middle of the output?
(2) What happens if the `randomize()` call is omitted?
A simple shell, continued

The second element for a shell is the `exec()` call:

```
exec(fname, arg0, arg1, ..., argN)
```

This call replaces the current process with an execution of the program named by `fname`, supplying the remaining parameters as arguments to the program.

A simple example: `(exec0.icn)`

```
procedure main()
    write("Ready to exec ls...")
    exec("/bin/ls", "ls", "-ld", "/")
    write("Done with exec...")
end
```

Execution:

```
% exec0
Ready to exec ls...
drwxr-xr-x  27 root  wheel   1024 Apr 13 16:56 /
%
```

Note that `exec()`'s `arg0` through `argN` corresponds to, e.g., `argv[0]` through `argv[N]` in a C program:

```
void main(int argc, char *argv[])
{
    ...
}
```
A simple shell, continued

As mentioned earlier, UNIX shells use a "fork and exec" sequence: When the user types a command to run, the shell forks and then uses an `exec()` call in the child to overlay the child process with the command of interest.

A very simple shell:

```plaintext
procedure main()
    while writes("Cmd? ") & cmdline := read() do {
        if (child := fork()) = 0 then {
            #
            # We're the child process. Split up
            # command line and exec it.
            w := split(cmdline)
            cmd := get(w)
            exec!(["/bin/"||cmd, cmd] ||| w)
        }
        else
            #
            # We're the parent. Wait for the child
            # to terminate before prompting again.
            wait(child)
        }
    end

Execution:

Cmd?  ls -ld /
drwxr-xr-x  27 root  wheel     1024 Apr 13  16:56 /
Cmd?  date
Mon Apr 21 04:13:29 MST 2003
Cmd?  wc /etc/passwd
     1462 3840 98991 /etc/passwd
Cmd?  wc </etc/passwd
   wc: cannot open </etc/passwd
Cmd?  who >out
   who: Cannot stat file '>
```
A simple shell—I/O redirection

UNIX shells allow standard input and standard output to be *redirected* with the `<` and `>` symbols.

Here is a shell command that runs `wc` on the class mailing list mailbox and redirects the output to the file `wc.output`

```
% wc < /home/cs451/mail > wc.output
```

The result:

```
% cat wc.output
 6257  31501  268989
```
IO redirection, continued

For reference:

```
% wc < /home/cs451/mail > wc.output
```

A cornerstone of redirection is that the `exec()` call replaces the current process with the execution of another program, but file descriptors are unaffected. For example, standard input (`&input`) in the original process is standard input in the replacement process.

Here is a program (redirl) that takes advantage of this carryover of file descriptors to mimic the shell command above:

```unicon
procedure main()

    infile := open("/home/cs451/mail")
    fdup(infile, &input)  # like &input := infile

    outfile := open("wc.output", "w")
    fdup(outfile, &output)

    exec("wc", "wc")
end
```

Execution:

```
% redirl
% cat wc.output
   6257   31501  268989
```

The `fdup(from, to)` function replaces the file descriptor associated with the file value `to` with the file descriptor associated with the file value `from`.
ish: A shell in Unicon

Here is ish, a rudimentary shell that provides I/O redirection and background processes (via &):

```unicon
procedure main()
    while line := (writes("ish -- "), read()) do {
        if *line = 0 then next
        w := split(line)
        cmd := get(w)

        background := if w[-1] = = "&" then { pull(w); 1} else &null

        if fork() = 0 then {
            pgmargs := []
            stdin := stdout := &null
            while arg := get(w) do {
                case arg[1] of {
                    ": stdin := arg[2:0]  # assume no space after '<' and '>'
                    ": stdout:= arg[2:0]
                    defaul: put(pgmargs, arg)
                }
            }
            if stdin then {
                stdin := open(stdin) | stop(stdin, ": Can’t open")
                fdup(stdin, &input)
            }
            if stdout then {
                stdout := open(stdout, "w") | stop(stdout, ": Can't open")
                fdup(stdout, &output)
            }
            exec!(cmd, cmd ||| pgmargs)
        } else {
            if /background then wait()
        }
    }
end
```
ish in operation

% ish
ish -- date
ish -- date >out
ish -- cat out
Wed Apr 23 23:33:54 MST 2003
ish -- wc <ish.icn
    38   115   832
ish --
ish -- du /usr >du.out &
ish -- wc du.out
    562  1124  21711 du.out
ish -- who
whm   tty1    Apr  6 23:50
whm   pts/0   Apr  6 23:52 (:0)
ish -- date
ish -- wc du.out
    1644  3288  64077 du.out
ish -- ^D
%

Some work remains:

ish --
ish -- ls *.icn
ls: *.icn: No such file or directory
ish --
ish -- ls | wc
ls: |: No such file or directory
ls: wc: No such file or directory
Pipes

A standard feature of UNIX shells is the ability to send the output from one program into the input of another program.

\`ls | wc\`
\`ls -t | grep -v ".icn$" | head -1\`

The supporting mechanism for this is called a pipe.

A pipe is an operating system mechanism that arranges for output written to a file descriptor to be available as input on another file descriptor.

Reads from a pipe will block until something is written to the other end. Writes to a pipe will block if a sufficient amount of already written data is still unread.

Unicon's `pipe()` function creates a pipe and returns a list of two file values: the first for reading and the second for writing:

\`][ pipe();\`
\`r := [file(pipe), file(pipe)]` (list)

A trivial example:

\`
procedure main(args)
    pipes := pipe()
    write(pipes[2], "Testing...")
    write(reverse(read(pipes[1])))
end
\`

Output:

`...gnitseT`
Pipes, continued

In most cases a pipe is used to send data between two processes.

In the following program a child process writes to a parent process at random intervals.

```unicon
procedure main()
    pipe_pair := pipe()
    if fork() = 0 then
        every i := 1 to 5 do {
            delay(?5000)
            write(pipe_pair[2], i)
        }
    end

    while line := read(pipe_pair[1]) do
        write("My child wrote to me! (", line, " at ", &clock, ")")
end
```

Output:

```
My child wrote to me! (1 at 21:14:04)
My child wrote to me! (2 at 21:14:06)
My child wrote to me! (3 at 21:14:08)
My child wrote to me! (4 at 21:14:10)
My child wrote to me! (5 at 21:14:13)
```
Pipes, continued

Consider a program that prompts for two commands and uses a pipe to connect the output of the first to the output of the second: (blank lines have been added...)

Pipe from? `ls`
Pipe to? `wc`
   118  118  917

Pipe from? `ls`
Pipe to? `grep fork`
fork
fork.icn
fork0
fork0.icn

Pipe from? `who`
Pipe to? `wc`
   71  355  2201

Pipe from? `iota 3`
Pipe to? `cat`
   1
   2
   3

Pipe from? `iota 3`
Pipe to? `tac`
   3
   2
   1
Pipes, continued

Here is the from/to piper:

```verbatim
procedure main()
    repeat {
        writes("Pipe from? ")
        from_cmd := split(read())
        writes("Pipe to? ")
        to_cmd := split(read())

        pipe_pair := pipe()
        if fork() = 0 then {
            fdup(pipe_pair[2], &output)
            close(pipe_pair[1])
            exec!([from_cmd[1]] || from_cmd)
            write("exec failed! (from)")
        }

        if fork() = 0 then {
            fdup(pipe_pair[1], &input)
            close(pipe_pair[2])
            exec!([to_cmd[1]] || to_cmd)
            write("exec failed! (to)")
        }

        close(pipe_pair[1])
        close(pipe_pair[2])

        wait()  # wait for both children to
t        wait()  # terminate
    }
end

Note that the close()s are needed to make it work.

How could we add piping to ish?
The `select()` function

The `select()` function allows a program to wait on input from any one of several input sources and, optionally, a delay time. It looks like this:

```unicon
select(file1, file2, ..., wait_time)
```

It returns when input is available on at least one of the files and/or the wait time (in milliseconds) has elapsed. The return value is a list of files on which input is available. If the list is empty, the wait time was exceeded.

In what situations is something like `select()` necessary?

`select()` allegedly works on files, network connections, pipes, and windows. The following example required a patch to the Unicon runtime system.
The `select()` function, continued

In this program a parent process forks three children and then waits to hear from each via a pipe.

```unicon
procedure main()
  cpipes :=[] # input side of pipes from children
  every c := !"ABC" do {
    pipe_pair := pipe()
    put(cpipes, pipe_pair[1])
    if fork() = 0 then {
      randomize()
      repeat {
        delay(?15000) # 15 seconds
        write(pipe_pair[2], c)
      }
    }
  }
  while files := select(cpipes[1], cpipes[2], cpipes[3], 3500) do { # should use select!...
    if *files ~= 0 then
      every f := !files do {
        line := read(f)
        write(line, " wrote to me at ", &clock)
      }
    else
      write("My kids never write...")
  }
end
```

Output:

<table>
<thead>
<tr>
<th>C wrote to me at 02:22:58</th>
<th>A wrote to me at 02:23:11</th>
</tr>
</thead>
<tbody>
<tr>
<td>My kids never write...</td>
<td>My kids never write...</td>
</tr>
<tr>
<td>B wrote to me at 02:23:04</td>
<td>B wrote to me at 02:23:18</td>
</tr>
<tr>
<td>A wrote to me at 02:23:04</td>
<td>C wrote to me at 02:23:19</td>
</tr>
<tr>
<td>My kids never write...</td>
<td>C wrote to me at 02:23:22</td>
</tr>
<tr>
<td>C wrote to me at 02:23:08</td>
<td></td>
</tr>
</tbody>
</table>
Defaulting and type conversion

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

\[
\text{parameter-name} : \text{conversion-procedure} : \text{default-value}
\]

Both \textit{conversion-procedure} and \textit{default-value} are optional.

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

\begin{verbatim}
class Rectangle(width, height)
    initially(w:integer, h:integer)
        width := w
        height := h
end
\end{verbatim}

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

\begin{verbatim}
][ 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
\end{verbatim}

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:

\[ \text{parameter-name} : \text{conversion-procedure} : \text{default-value} \]

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:

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

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

Usage:

```unicon
][  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:

```unicon
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()**:

```unicon
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 any structure written with `xencode()`:

```unicon
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:

```bash
% 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?