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.

```
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.

```
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:

```
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?

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:

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

are translated into this Icon code:

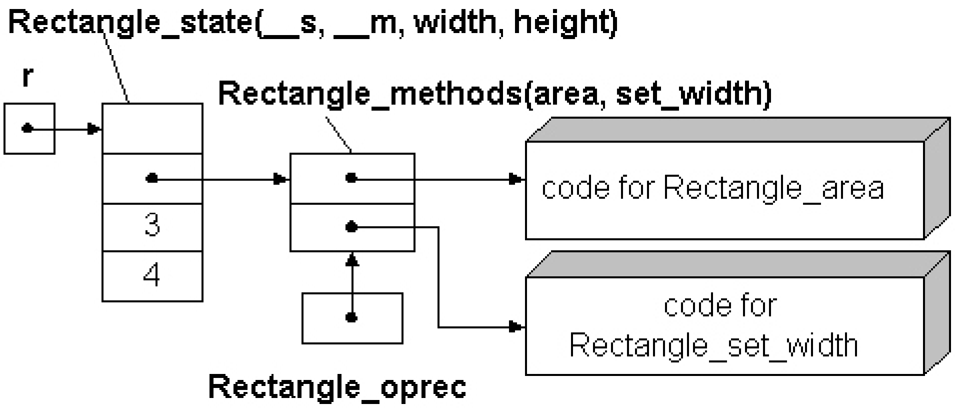
```
procedure Rectangle_area(self)
    return self.width * self.height
end

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

Behind the scenes in Unicon, continued

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

```
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:


Rectangle_state(__s, __m, width, height)
r
Rectangle_methods(area, set_width)
3
4
code for Rectangle_area
code for Rectangle_set_width
Rectangle_oprec

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`:

Field name	*Description*
`dev`	ID of device containing the file
`ino`	Inode number
`mode`	File mode (e.g. protections)
`nlink`	Number of links

`uid,`
`gid`
	User-id and group-id

`size`	Size of the file in bytes
`atime`	Time of last access
`mtime`	Time of last modification
`ctime`	Time of last inode change
`symlink`	If a symbolic link, the name linked to.

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 04:45 v1
-r-xr-xr-x   1 whm      cs451        3730 Feb 12 22:33 day
-r-xr-xr-x   1 whm      cs451        3450 Feb 12 21:54 mcycle
-r--r--r--   1 whm      cs451         148 Feb 12 21:54 mcycle.1
-r--r--r--   1 whm      cs451         701 Feb 12 21:54 mcycle.2
-r-xr-xr-x   1 whm      dept        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`)

bysize.icn

```
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:

```
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 '>out'
```