Name:____

CSc 451, Spring 2003
Final Examination
May 15, 2003

READ THIS FIRST

Fill in your name above. Do not turn this page until you are told to begin.

DO NOT use your own paper. If you run out of room, write on the back of the page.

If you have a question that can be safely resolved by making an assumption, simply write down that assumption and proceed. Examples:

> "Assuming `reverse(s)` reverses a string"
> "Assuming `*t` returns the number of keys in table `t`"

If you have a question you wish to ask, raise your hand and the instructor will come to you. DO NOT leave your seat.

You may use all elements of Icon.

You may use Icon procedures that have appeared on the slides, or been presented in class, or been mentioned in e-mail, or that have appeared on an assignment this semester, or that are mentioned in this exam.

There are no deductions for poor style. Anything that works and meets all problem-specific restrictions will be worth full credit, but try to keep your solutions brief to save time.

You need not include any explanation in an answer if you are confident it is correct. However, if an answer is incorrect, any accompanying explanation may help you earn partial credit.

If you are unsure about the form or operation of a language construct that is central to a problem's solution, you are strongly encouraged to ask the instructor about it. If you're completely puzzled on a problem, please ask for a hint. Try to avoid leaving a problem completely blank—that will certainly earn no credit.

This is a one hundred minute exam with a total of 100 points and 5 possible points of extra credit. There are eleven regular problems and an extra credit section with five questions.

When you have completed the exam, enter your name on the exam sign-out log and then hand your exam to the instructor.

When told to begin, double-check that your name is at the top of this page, and then put your initials in the lower right hand corner of each page, being sure to check that you have all fifteen pages.

Problem 1: (3 points)

In some cases it would be nice to have a convenient syntax to initialize some number of table entries when a table is created. Here is a proposed addition to Icon:

> If the argument to `table()` is a list of N elements where N >= 2 and even, then the elements are assumed to be [key1, value1, key2, value2, ...] and those key/value pairs are added to the table.
>
> For example,
>
> ```
> t := table(split("x 1 y 2 z 3"))
> ```
>
> would be equivalent to
>
> ```
> t := table()
> t["x"] := "1"
> t["y"] := "2"
> t["z"] := "3"
> ```

Comment on the merit of this proposed addition and if you see problems with it, make a counterproposal that is superior and argue its merit.

Problem 2: (2 points)

Consider the idea that `+` be used instead of '`||`' for string concatenation, and instead of '`|||`' for list concatenation. For example, `"xyz" + "123"` would produce the string `"xyz123"` and `[1,2] + [3,4]` would produce the list `[1,2,3,4]`. Cite one advantage and one disadvantage of this broader meaning for the `+` operator. Ignore the problem of invalidating some existing code.

Problem 3: (2 points)

Given this Unicon procedure,

```
procedure f(s1:split, s2:2)
     return *s1*s2
end
```

What does `f("just testing")` return?

Problem 4: (3 points)

Consider this statement:

```
if (c == "x") | (c == "X") then ...
```

Show three different ways to perform the same comparison that are more concise.

Problem 5: (4 points)

Write a procedure `span(L)` that suspends the smallest value in `L` and then suspends the largest value in `L`. Assume that `L` is a list of integers. `span(L)` fails if L is empty.

Examples:

```
][ .every span([5,1,3,1,9,-10]);
   -10  (integer)
   9  (integer)

][ .every span([3]);
   3  (integer)
   3  (integer)

][ span([ ]);
Failure

][ span([5,1,3,1,9]);
   r := 1  (integer)

][
```

Problem 6: (15 points)

Write a program `lcount` that accepts one or more file names as command line arguments and prints the number of lines that each file contains. If no command line arguments are specified, lines on standard input are counted instead.

Examples: (For clarity, a blank line has been added after each.)

```
% lcount short.icn expand.icn toby.icn unique.1
short.icn    1 lines
expand.icn  19 lines
toby.icn     7 lines
unique.1    10 lines

% lcount x maybe_its_time_to_put_data_IN_the_file
x                                      140 lines
maybe_its_time_to_put_data_IN_the_file 260 lines

% lcount
a
test
^D (control-D)
<stdin>   2 lines

% lcount < short.icn
<stdin>   1 lines
%
```

The file name should be left-justified in a field wide enough to accommodate the name of the longest input file. The line count should be right-justified in a three-character field.

Note that if standard input is processed the name "`<stdin>`" should be shown

If a file can't be opened, an error message should be printed and execution terminated.

(Space for solution on next page)

SPACE FOR SOLUTION FOR PROBLEM 6

Problem 7: (8 points)

Characters in a URL can be specified using a two-digit hexadecimal code preceded by a percent sign. For example, instead of "`http://www.google.com`" one might use "`http://www%2egoogle%2eco%6d`".

Write a procedure `urlhex(s)` that returns a copy of the string `s` with any such hexadecimal specifications converted to the corresponding characters. Examples:

```
][ urlhex("http://www%2egoogle%2eco%6d");
   r := "http://www.google.com"  (string)

][ urlhex("testing");
   r := "testing"  (string)
```

IMPORTANT: The procedure fails if a percent sign in the argument is not followed by two valid hexadecimal digits:

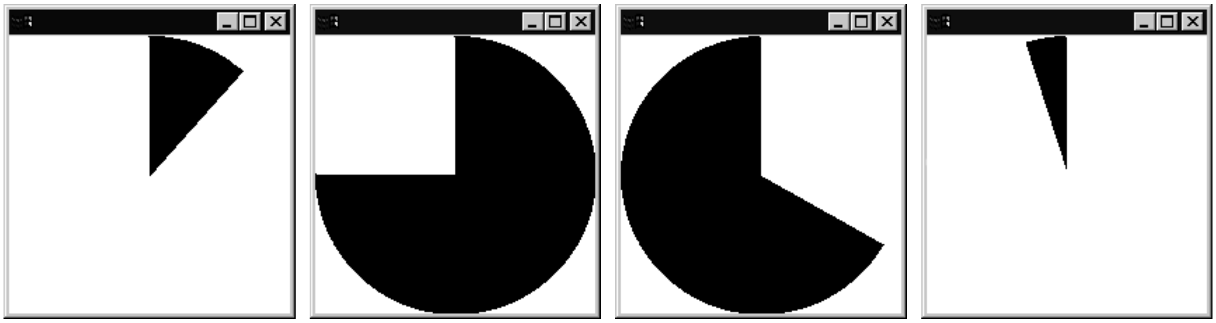
```
][ urlhex("ab%2g");
Failure

][ urlhex("ab%2");
Failure
```

Hint:

```
][ char(integer("16r61"));
   r := "a"  (string)
```

Problem 8: (12 points)

In this problem you are to write a graphical program that repeatedly sweeps out a circle and then erases it. It takes one minute to sweep out the circle and one minute to erase it. There should be an update once per second. Here are screen shots at various points in time:



`7 seconds` `45 seconds` `80 seconds` `157 seconds`

At 60, 180, 240, ... seconds a full circle is present. At 120, 240, 360, ... seconds no trace of the circle is visible. A "q" exits the program in no more than one second. All other events are ignored.

You'll want to use `FillCircle(x, y, r, start, extent)`. Remember that `start` and `extent` are measured in radians; a starting position of zero is at 3 o'clock; and that a positive extent goes in a clockwise direction. `WDelay(N)` pauses execution for N milliseconds.

Problem 9: (20 points)

In this problem you are to implement four classes in Unicon. The first is named `Eater`. Instances of `Eater` are initialized with an integer that is the `Eater`'s comfortable capacity, expressed in food units:

```
e1 := Eater(10)
```

An `Eater` can be told to eat something:

```
e1.eat("apple")
e1.eat("Big Mac")
```

A food is considered to have as many units as characters in its name. The two feedings above represent a total of twelve units: 5 for the apple and 7 for the Big Mac.

If an `Eater` consumes more than its capacity, it then burps to make room for more food. In the above case, `e1.eat("Big Mac")` would produce this output as a side effect:

```
Burp!
```

Each burp reduces the volume currently retained in the `Eater` by the capacity of the `Eater`. After the burp, `e1` still has two units of food.

A large feeding may produce several burps:

```
e1.eat("Big Ed’s Belly Buster Special")  # 29 characters
```

Output:

```
Burp!
Burp!
Burp!
```

Note that three burps are produced because `e1` had two units remaining after its first burp.

An `Eater` may be asked what it has had to eat. `e1.eaten()` produces this output:

```
Apple
Big Mac
Big Ed’s Belly Buster Special
```

The second class to implement is `Food`. `Food` and has two methods: `units()` and `what()`. `units()` returns the number of units for the food. `what()` returns a description of the food.

Implement two subclasses of `Food`: `Burger` and `Fries`. A `Burger` has 25 units of food and `Fries` have 15 units. Example:

```
][ b := Burger();

][ b.units();
   r := 25  (integer)

][ b.what();
   r := "Burger"  (string)
```

Here's an example with subclasses of `Food`:

```
e3 := Eater(20)
e3.eat(Burger())
e3.eat(Fries())
e3.eat("Nachos")
e3.eat(Burger())
write("e3 has had...")
e3.eaten()
```

Output:

```
Burp!
Burp!
Burp!
e3 has had...
Burger
Fries
Nachos
Burger
```

Here's a capsule summary of what you must implement:

```
Eater            Initialize with capacity, an integer
   eat(x)        Consume x, which may be a string or a subclass of
                 Food.  Burp until relieved.
   eaten()       Print what's been eaten, one item per line

Food
   units()       Return the number of units for this food
   what()        Return the name of this food

Burger           Subclass of Food, 25 units
Fries            Subclass of Food, 15 units
```

There are no specific requirements for the constructor for `Food`—choose something that works with your implementation of `Burger` and `Fries`.

SPACE FOR SOLUTION FOR PROBLEM 9

Problem 10: (3 points)

Comment intelligently on this statement: *In Icon, variables are implicitly declared upon their first use. For example, if the variable* `x` *has not been previously used, the statement* `x := 3` *declares that the type of the variable* `x` *is an integer, just like '*`int x = 3;`*' in Java.*

Problem 11: (4 points each; 28 points total)

<u>IMPORTANT: ANSWER ONLY SEVEN OF THE FOLLOWING THIRTEEN QUESTIONS</u>. If you answer more than seven, the first seven will be graded. Be sure to clearly strike through any answers that you start on but then abandon.

Each of following questions is based on one proposed change to Icon (or Unicon) by each student who submitted assignment 1.

Proposed change: *Don't allow the names of built-in Icon functions, such as* `pos`*, to be used as variables.*

Question: Unintentional assignments to built-in functions can certainly cause some headaches but in some cases it's very handy. Present a set of language additions/modifications that simultaneously and conveniently accommodates these three different schools of thought: (1) I like things the way they are. (2) Never let me assign to the name of a built-in procedure. (3) Don't let me assign to a built-in unless I specifically indicate I want to.

Proposed change: `write()` *should return the full string it printed, not just the last argument.*

Question: What would be a negative impact of this change, and why?

Proposed change: *Icon has too many operators. I cannot think of any other language with so many operators to remember. At some point it's more useful to have functions than to make the programmer use a cheat-sheet.*

Question: Specify four operators that it would be good to replace with functions and suggest a function name for each that's easier to remember than the symbolic operator.

Proposed change: *When you try to assign a value outside a list's boundaries, the list should automatically expand itself and fill any gaps created with nulls. For example (`L[3] := "c"` s hould create a list L of `[null, null, "c"]`).*

Question: For the expression `L[N]`, what's a value of N that would pose a problem with the above rule, and why? (Hint: For full credit you must think of an issue that's at least as good as the one the instructor has in mind.)

Proposed change: *The `repeat` and `until-do` loops should not be included in the language. Although they are considered "syntactic sugar", their functionality can be modeled using the regular while loop with minor changes.*

Question: Present a strong argument that `until-do` is a worthwhile element of the language.

Proposed change: *Unicode support, i.e., support for 16-bit characters should be added to Icon. Characters shouldn't be limited to the English character set.*

Question: What element of the language would be most severely impacted, in terms of both speed and space, by adding Unicode support. Why?

Proposed change: *The `*` operator shouldn't show the number of results already generated by a co-expression, instead it should be the total number of results that the co-expression will generate.*

Question: Cite a co-expression with a finite result sequence for which the number of results it will produce can not be predicted.

Proposed change: *Icon permits the use of the* `insert`, `delete` *and* `member` *functions for sets and tables.* `delete` *should be supported for lists as well.*

Question: With the idea of being able to delete from lists in mind, what's an additional aspect of the behavior of `delete` that would need to be specified?

Proposed change: *Any type determination that can be made at compile-time should be made to improve runtime performance.*

Question: Give a specific example of how compile-time type determination can be used to reduce execution time.

Proposed change: `member()`, `insert()`, *and* `delete()` *should work on both sets and csets.*

Question: That might create a pitfall for a newcomer to Icon. What is it?

Proposed change: *You should be able to specify an initializing value for a global variable. For example,* `global whitespace := ' \t'`

Question: Depending on the implementation of this feature, a potential inconsistency could be created. What is that potential inconsistency?

Proposed change: *Give lists and sets value semantics.*

Question: What's meant by that statement? (Be sure to include an example involving some code.)

Proposed change: *Lists should be invocable. The invocation `[proc, a, b](c, d)` would be equivalent to `proc(a, b, c, d)`, and `[p]()` would be like `p()`, for example.*

There is a notion of *partial evaluation* of functions wherein a function can be supplied some of its arguments, thereby creating a new function that if called with the balance of its arguments produces a final result. Here is what could be done in Icon, with the above-proposed change:

```
][ replx := partial(repl, "x");
   r := <not shown>

][ x10 := replx(10);
   r := "xxxxxxxxxx"  (string)
```

`replx` references a value that represents a partially evaluated form of `repl`, which has had its first argument supplied. The call `replx(10)` invokes that partially evaluated form, producing the same result as if `repl("x", 10)` had been called.

Question: Assuming the presence of list invocation as described above, provide an implementation of `partial` that would produce a suitable value for `replx`.

EXTRA CREDIT SECTION

(1 point) List the last names of ten other students in this class. Use of phonetic spelling is acceptable.

(1 point) To the best of the instructor's knowledge, the last time a class at the University of Arizona covered Icon's graphics facilities was in the mid-1990s. In general, students then had a very positive response to the graphics facilities. The response this semester, particularly to VIB and vidgets, was not so warm. Why?

(1 point) Write a routine `which(x)` that returns "`string`" or "`list`", depending on whether `x` is a string or list. You may not use `type()` or `[Ii]mage()`. You may assume that `x` is either a string or a list.

(1 point) The text list vidget provides no way to indicate that there was a double-click on a item. Describe a technique that would allow a program to respond to a double-click on a text list item. The text list vidget must be used as-is. (If you tried to do this on your project but it didn't work out, briefly describe what you tried.)

(1 point) Write a SNOBOL4 program that reads lines from standard input and prints the first and last line on standard output. Recall that referencing the variable `INPUT` causes a line to be read and that assigning a value to the variable `OUTPUT` causes a line to be written.