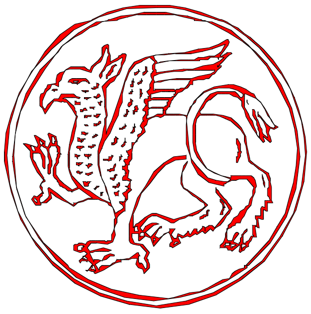
The Generator

In this issue:



"I know what I have witnessed. Now it is your turn. Prepare yourself for a journey into a world where each new step may give you a better understanding of your own reality."

VOL 2. NO 2.

DECEMBER MMVI.

Statement of Purpose

The Generator is an international, non-for-profit journal devoted to the use of the Unicon programming language and its predecessor and subset, the Icon programming language.
The Generator can be freely redistributed in its complete and unchanged form.

Printing Instructions

The Generator is designed to be printed on the both sides of the paper. It is published in Letter format; we hope our readers can manage to print it on A4 paper when needed, and welcome suggestions. In some situations, the printing options *Auto-rotate and Center* and *Fit to paper* may be appropriate.

Call for Papers

The Generator publishes a wide range of articles: papers, reviews, notes, reports etc. All articles contain some amount of the previously unpublished material; an exception is material previously published on less formal ways (preprints, mailing list and newsgroups posts etc.)
No particular article style is preferred.
Code should follow Icon and Unicon Projects' standard formatting conventions.
All submitted articles are reviewed.
The author permits unlimited publishing of the submitted article in The Generator, including any printed/bound issues, reprints, and collections.
Copyright of the articles is not transferred to The Generator.

Editorial Board

Typesetting

Typesetting is done by authors, reviewers and the publisher of The Generator.
The illustrations: Monsters of Stone, Omega Font Labs, <http://www.moorstation.org/typoasis/designers/omega/omega.htm> and DBL Corners, House of Lime, <http://www.houseoflime.com>.
The fonts: Weatherly Systems Inc. Ramona, Bitstream De Vinne and Pica10 and Corel *Frankenstein*.

Publisher

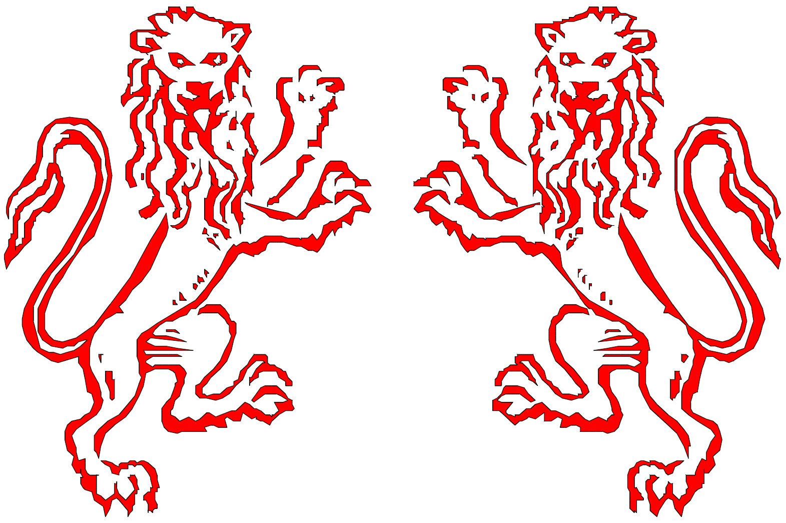
Clinton Jeffery, Moscow, Idaho USA.

Ralph Griswold's Legacy

Ralph Griswold passed away this year. His work on SNOBOL with industry colleagues is what Ralph is best known for, but I suspect that his greatest long-term legacy will end up being the impact he had on many students. My impression, which others may correct, is that Ralph was not the sole inventor, nor even a primary coder for the Icon language, he chose the more difficult path of challenging students to rise to the occasion. Sometimes he was happier with the results than at other times, but in all cases the students learned more this way.

Ralph's legacy vis a vis Unicon is also interesting. He commissioned early work on Unicon's predecessor Idol, specifically funding its initial distribution and the writing of the Idol technical report. However, Ralph had no particular interest or need for objects, classes, or packages: he mostly wrote Icon programs that were seldom larger than a few hundred lines. When Idol merged with Shamim Mohamed's Unicon POSIX facilities and Federico Balbi's ODBC extensions, Ralph specifically objected to calling the result Icon-2, not wanting any misunderstanding about whether Icon Project had been involved (it had not, other than contributing 90% of the code by means of Icon's public domain release). So indirectly, Ralph helped choose our name, and the name change incidentally freed us from the chains of a publishing contract for the Unicon book, propelling us towards an open documentation license.

Although Icon has influenced many popular languages, at this writing no language other than Icon and Unicon have really gotten generators and goal-directed evaluation “right”. Whether Ralph would be happy with where Unicon has taken his language family and where it is heading is anyone's guess. From this editor's perspective, the important thing about Unicon is to make Ralph's language family useful to more people in a broader range of application domains than Icon serves. This includes larger scale, higher performance applications with extensive input/output requirements that go beyond Icon's string and list processing, and text file manipulation roots. Icon continues to serve as an example to more mainstream languages such as Python. The better Unicon manages to become, the more relevant that example, perhaps the more that mainstream languages may understand and adopt the benefits of generators and especially, goal-directed evaluation.

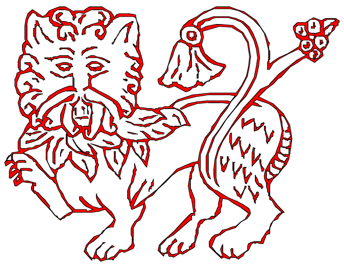


Unicon Project Moves North

Unicon Project is moving to Moscow, Idaho, where Clinton Jeffery and his team are joining the University of Idaho. Unicon's new home at the edge of the Rocky Mountains places it at an institution where Unicon research and development can advance. Aside from allowing Jeffery more time for Unicon-related research and support, Idaho appears to be a better place for the doctoral students who create some of the most important advances in the implementation.

Of course, Unicon's hosting on Source Forge is not affected by such moves. Source Forge has had difficulty serving up large files from its web servers, so a virtual move of the project to a new host is not inconceivable, but will be done only as a last resort if no alternative is possible.

Those of you with a sense of humor will no doubt want to refer to this issue as “the co-expressions issue” and then look back at preceding issues and wonder if The Generator is mainly devoted to the study of co-expressions! We will keep running articles on co-expressions whenever they are available, but I expect The Generator to have many different themes in future issues.



A Unicon Shell

Art Eschenlauer *eschen@alumni.princeton.edu*

Abstract

A proof-of-concept implementation of a shell for Unicon tasks is described. The shell permits new solutions to be composed from previously translated Unicon programs. The shell can read and execute a script that defines a new solution without invoking the Unicon translator. Compatible tasks are ordinary Unicon programs in every way except that they use the Stream interface in lieu of resumption expressions when transferring control of execution to other tasks in the solution. The Stream interface also provides uniform methods for sequentially exchanging data values with co-expressions, lists, and files.

Introduction

As computer programs become increasingly complex, it is necessary to develop strategies to manage their complexity. Object-oriented methodology is recommended to organize and encapsulate data when programs grow beyond 10,000 lines or so. The "Unix philosophy" [1] takes a different approach to building complex systems, which has been summarized (by Doug McIlroy and Eric S. Raymond respectively) as "Write programs that do one thing and do it well. Write programs to work together." or "Write simple parts connected by clean interfaces." In other words, rather than composing complex programs, compose solutions to complex problems by creating combinations of simple, well-tested programs.

In the Unix user's view of the world, classic "simple" tours-de-force are invoked from an installation's bin directories. The programs typically read and write text streams, so that the user can use pipes to combine them into powerful solutions, with each program running in its own process space. This is so effective that the user frequently can solve a problem by combining existing programs rather than resorting to writing any new programs. Creating solutions this way is one application of the "pipes and filters" design pattern [2].

Using pipes to compose multi-process solutions from existing programs has worked very well for more than 30 years; however, it does force programs to serialize data into (usually text-based) streams in order to pass them to other programs, regardless of whether two collaborating programs are executed on the same CPU or on different continents. Although an extremely persuasive argument can be made for always using plain text serialization as the interface mechanism, other inter-process communication [3] mechanisms have been employed, particularly when coupling several processes that are parts of a tightly integrated system such as a database management system or a web server.

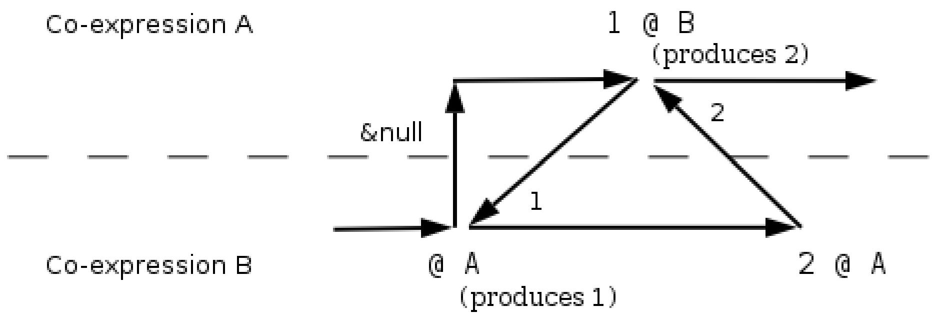
One alternative to partitioning each processing step into a separate process is to put each step into one of several coroutines [4], all of which are executed within a single thread of execution. The process "resumes" each coroutine (i.e., makes it the active coroutine) when the next datum needed by that coroutine becomes available. A coroutine resembles a subroutine in that it is made active by other routines. It differs from a subroutine in one important respect: A subroutine returns, abandons its state, and (the next time that it is invoked) starts from its entry point; a coroutine, by contrast, suspends execution, maintains its state, and (the next time it is resumed) continues execution from the point at which it suspended.

Icon co-expressions provide a framework for straightforward implementation of coroutines [5,6]. Because each co-expression is created with its own execution stack and its own copies of the local variables of the procedure in it was created, it can maintain its state and resume execution from the point at which it yielded control. When one co-expression enters a *resumption expression*, it yields control of execution to another co-expression. (A co-expression that has previously yielded control of execution is said to be *resumed* when another co-expression yields to it. On the other hand, a co-expression beginning execution from its entry point, without having previously yielded control of execution, is said to have been *activated*. A resumption expression may either activate or resume another co-expression.)

When a co-expression enters a resumption expression, it transmits a data value to the co-expression that it is resuming, but if no value is supplied in the resumption expression, the null value is transmitted. Transmitted data values may be of any Icon datatype. A resumption expression not only transmits data and yields control of execution, but it also produces whatever value is later transmitted when another co-expression resumes the co-expression in which the resumption expression appears; hence, the only way that a co-expression can "receive" a transmitted data value is if it has previously yielded control via a resumption expression (hence, any value transmitted during activation of a co-expression is discarded). For an illustration, consider this scenario:

1. Co-expression B activates co-expression A in the resumption expression `@ A`
2. Co-expression A transmits the value 1 to co-expression B in the resumption expression `1 @ B`
3. Co-expression B transmits the value 2 to co-expression A in the resumption expression `2 @ A`

The resumption expression `@ A` produces the value 1, and `1 @ B` produces the value 2:



Co-expressions provide the basis for implementation of MT Icon, the multitasking Icon interpreter [7] that is incorporated into the Unicon programming language [8,9]. MT Icon permits a Unicon program to load and execute several previously translated Unicon programs within the same interpreter thread. Each loaded program, referred to as a “task”, begins execution in its own co-expression, starting from its "main" entry point. MT Icon is a co-operative, single-threaded multitasking system; the tasks switch into and out of the thread of execution by the Icon co-expression activation/resumption mechanism. MT Icon and Unicon do not support multiple threads per process; the interpreter runs in a single thread and executes only one task at a time. Communication among the several tasks, therefore, becomes a matter of intra-process (and intra-thread) communication, and different tasks can safely share references to the same data in memory. There are several well-defined interfaces by which one task may obtain data values or references to data structures (such as tables and lists) from another task:

- As with ordinary co-expression resumption, data values or references are transmitted when one task resumes another.
- When a parent task loads a child task, MT Icon permits the former to pass to the latter a list of arguments that may have any Unicon type, e.g., an argument may be a non-string value and can even be a reference to the &main co-expression of another task.
- One task may obtain a reference to a variable within another task using the `variable( )` function.

This article describes a proof-of-concept Unicon implementation of a "shell" for Unicon. This shell program:

1. Takes as input a text script that defines the composition of a solution from several tasks
2. Specifies which tasks should be activated, in what order.
3. Uses the `load( )` function to load each task from a previously interpreted Unicon program.
4. Establish a coroutine for each task.
5. Activates tasks as specified in the input script.

The shell relies upon a "Stream" interface that:

- Ensures that the sequence of values that a consumer receives from a producer does not depend on whether the consumer or the producer is activated first.
- Provides a single set of methods for sequentially exchanging data values with lists, files, and co-expressions.

Use Cases

The principal value of a shell is that it permits the user to execute a complex solution assembled from previously created and tested components. In the context of this report, a solution is realized by an input script, and a component of a solution is realized by a translated Unicon program that can be loaded into the Unicon interpreter as a task.

Use Case: Run Solution

1. The user invokes the shell program with an input script.
Extension: If no script exists defining a desired solution, the user needs to ***Create Solution****.*

Use Case: Create Solution

1. The user envisions a solution composed of several components.
Extension: If the solution is a complex combination of components, the user may benefit from drawing a graph describing the relationships among the components of the solution.

2. For each component of the solution, the user selects a previously translated Unicon program that the shell may load and execute as a task.
Extension: If no program exists for a task, the user must ***Create, Translate, and Test Program****.*
3. The user determines the argument list for invoking each task.
4. The user writes and tests the shell script.

Use Case: Create, Translate, and Test Program

1. The user writes a conventional Unicon program.
   - At any point where the user plans for the program to transfer *both* a data value *and* the control of execution to another task, the user applies the `Get( )` or `Put( )` method of the Stream interface to access the task.
2. At points where the user plans for the program to exchange data values with a file or list (while retaining control of execution), the user may choose to apply appropriate methods of the Stream interface to access the file or list reference.
3. The user invokes the Unicon translator and tests the program.

Design Objectives

The Unicon Shell is organized in two components, a shell interpreter and an underlying streams interface. The following design objectives guided the implementation of shell.icn and Stream.icn:

shell.icn

- shell.icn should interpret scripts, so that the Unicon translator is not required to execute a new solution composed from previously translated Unicon programs.
- shell.icn should load tasks and pass co-expression references (for other tasks) as arguments to tasks. However, shell.icn should *not* be responsible for determining the role that a task plays in producer-consumer or client-service relationships. The burden of determining that role is placed on the programmer defining and coding the task itself.
- shell.icn should permit explicit specification of lists that several tasks need to access, to ease implementation of inter-task queuing of data values.
- shell.icn should provide a mechanism whereby two child tasks do not go into "infinite failure" (where, alternately, each of the two tasks' co-expressions is resumed by and transmits failure to the other). This mechanism should not place an undue burden on how the child tasks are coded.

Stream.icn

- Stream.icn should provide a common interface for exchanging data with files, co-expressions, and lists.
- Operations that are essentially equivalent should be invocable by this interface in the same way regardless of whether a file, list, or co-expression is encapsulated by the Stream. Stream.icn should thereby provide a mechanism whereby a producer or consumer program can first be tested using Streams encapsulating the standard input and standard output files and then, without modification, the program can be run with shell.icn, using Streams encapsulating co-expressions.
- Stream.icn should implement a mechanism to permit a producer co-expression to pass the same sequence of values to a consumer co-expression, regardless of whether the producer or the consumer is activated first.

- Stream.icn should implement routines to ease implementation of switching between a "client" task and a "service" task.

Stream.icn - A Common Interface to Co-expressions, Files, and Lists

The `Stream( )` constructor produces instances of three classes: class Streamf for files, class StreamC for co-expressions, and class StreamL for lists. Because these classes present a common interface, programs can be written to use instances of these classes without the need to distinguish which class an object instantiates. However, although all three classes present the same interface (i.e., have the same methods defined), the `Push( )` and `Pull( )` methods succeed only for instances of StreamL.

The class produced by `Stream( )` depends on the arguments passed when `Stream( )` is invoked:

Invocation	*Produces*	*Behavior of* ***`Stream( )`*** *Constructor*
`Stream(f)`	Streamf	When an open file is the argument, `Stream( )` creates an instance of the file-encapsulating class, and produces that instance as the result. This instance can be used to manage this previously opened file.
`Stream(s1,s2,...)`	Streamf	When the arguments are the same as the arguments for `open( )`, `Stream( )` typically opens the file, creates an instance of the file-encapsulating class, and produces that instance as the result; this instance can be used to manage the newly opened file. However, when the filename is "-" and the filemode is "r" or "w", behavior is the same as for `Stream(&input)` or `Stream(&output)`, respectively.
`Stream(L)`	StreamL	When the argument is a list, `Stream( )` creates and produces an instance of the list-encapsulating class.
`Stream(C,s2)`	StreamC	When a co-expression reference is the first argument and a filemode ("r" or "w") is the second argument, `Stream( )` creates and produces an instance of the co-expression-encapsulating class. A program designed to act as a producer should use the "w" filemode when creating a StreamC instance to manage communication with a consumer task; a consumer should use the "r" filemode. A program designed to act as a client should use the "w" filemode when creating a StreamC instance to manage communication with a service task.

where f = file, s1 = filename string, s2 = filemode string, L = list, C = co-expression

Each class implements the following interface:

Method	*L*	*f*	*C*	*Behavior*
`Get(x?):x`	OK	OK	OK	Produces the value produced by `get(L)`, `read(f)`, or

				`(x@C)\1`. If the Stream encapsulates a co-expression, then `Get( )` can transmit one value (the argument of `Get(x)`) and produce another.
`Put(x+)` : L	OK	OK	OK	For StreamC, transmits each argument (beginning with the first) to the underlying co-expression, and produces a list of the results produced by the co-expression resumptions. For StreamL, puts each argument (beginning with the first) onto the end of the underlying list, and produces a list that consists of the arguments in the order that they appeared in the argument list. For Streamf, writes each argument (beginning with the first) to the underlying file, and produces a list that consists of the arguments in the order that they appeared in the argument list.
`Pull( )` : x	OK	fails	fails	Produces the value that would be produced by `pull(L)`.
`Push(x+)` : L	OK	fails	fails	Pushes each argument (beginning with the first) onto the beginning of the underlying list, e.g., if three arguments are provided, then after the operation is complete, the list will begin with the third argument. Produces a list that consists of the arguments in the order that they appeared in the argument list.
`Select(i)` : i | n	OK	OK	OK	Produces the null value (for Streamf or StreamC) or the length of the list (for StreamL) if the select succeeds, i.e., if `Get( )` is expected to produce a value. Fails if `Get( )` is expected to fail. For StreamC, the expectation is based on whether the last resumption of the co-expression succeeded. The argument specifies the timeout in milliseconds for an encapsulated file.
`Flush( )` : f | n	OK	OK	OK	Produces the result that `flush( )` produces for a file. Produces a null value for other classes.
`Close( )` : f | n	OK	OK	OK	For Streamf, closes the underlying file, and produces the result that `close( )` produces for the file. For StreamL, replaces the underlying list with an empty list, and produces the null value. For StreamC, transmits failure to the underlying co-expression if and only if filemode was "w" when object was constructed. Produces the null value. May be called repeatedly.
`Type( )` : s	OK	OK	OK	Produces a string ("file", "list", or "co-expression") corresponding to the underlying datatype of the instance of the StreamC class.
`Data( )` : L | f | C	OK	OK	OK	Produces the co-expression, list, or file underlying the instance of the StreamC class.

where L = list, f = file, C = co-expression, s = string, i = integer, n = &null, | = or,
x = any value, x? = 0 or 1 argument of any value, x+ = 1 or more arguments of any value

StreamC has an important additional feature. Because a co-expression cannot receive a transmitted value until after it has been activated and has yielded control of execution, it would ordinarily be necessary to activate the consumer in a producer-consumer relationship before resuming the producer (or the service in a client-service relationship). If the StreamC class' constructor method is invoked with a "w" filemode, it ensures that the co-expression that it manages has been activated before transmitting a value to it. StreamC tracks this via the set referenced by the global variable **StreamC_activated**; this set may be created in another task.

To aid in tracing transfer of control between tasks, Stream.icn defines global variables **StreamC_trace** and **trace**. To trace execution of StreamC code (without modifying and re-translating Stream), a task can assign a string value to **StreamC_trace**; this is demonstrated in the "Filter Pattern" example below. If **StreamC_trace** is not null and **trace** is null when the StreamC constructor is invoked, the constructor assigns it the value for the built-in function `write( )`. Programs presented in the appendix adhere to an informal standard of writing programs to accept an optional first argument of "-t"; if they receive this argument, they assign a string value to **StreamC_trace** (to turn on this tracing).

The reason that no Stream interface was implemented for strings is that Unicon (and Icon) strings are not mutable structures; all string operations produce new strings in contrast to modifying existing strings. Whether a Stream interface for strings should be implemented raises philosophical questions (e.g., is it sensible and similar?) and semantic questions (e.g., should the `Get( )` method produce substrings, and, if so, should it use newline characters in the string as a separator?).

Caution: Beware the effect of backtracking in expressions that invoke the `Get( )`, `Put( )`, `Pull( )`, `Push( )`, *and* `Close( )` *methods.* Theoretically, if one of those methods is resumed for an additional value, its effects on managed co-expressions and files cannot be reversed. Therefore, for consistency, the interface is designed so that all methods produce exactly one value on success, i.e., they are not generator functions. It is easy, however, to write expressions mistakenly that result in multiple invocations of these methods. For example:

```
every ( s := ( (1 to 3) || mystream.Get( ) ) ) do {
  write( s )
}
```

will get three values from mystream, which is correct but which may not be what the programmer is expecting to happen.

shell.icn Launches Collaborative Multiprogram Solutions

The shell.icn program interprets input scripts defining solutions, each of which may be composed of several tasks. Each task is defined by an "icode file", i.e., a translated Unicon program. The tasks are loaded and activated as specified in the script. The tasks collaborate via the co-expression switching mechanism; shell.icn monitors the task-switching events and intervenes to prevent "infinite loops of failure".

shell.icn - Usage

shell.icn is invoked as:

```
shell [options] {arguments}
```

where options are:

```
-t       - trace execution of the script interpreter
-i file - read script to be interpreted from file
-p prog - invoke prog as preprocessor
(option -p overrides option -i)
```

If neither the -i option nor the -p option is specified, the script is read from the standard input. See "The Preprocessor Pattern and a Command Line Interface Program" below for an example of using the -p option.

The use of arguments is demonstrated in the "Recursive Invocation and Argument Substitution" example below.

shell.icn - Input Script Format and Grammar

The goal of this project was to produce a working model of a Unicon shell. The grammar was chosen arbitrarily, with the objectives of balancing simplicity of implementation and simplicity of use (as subjectively evaluated by the author) to use for experimentation and to serve as a starting point for discussion. No assertion is made that the grammar chosen is the most appropriate, expressive, or useful for shell scripts; there may well be room for improvement. Here is a simple example script:

```
# Assign to the CONS symbol the task defined by
#   the icode file "consumer", invoked with two arguments,
#   PROD (another task) and the string "-".
#   consumer.icn is coded to open its first argument as a
#   Stream with filemode "r" and to open its second
#   with filemode "w".

CONS    := consumer  PROD -  # task_declaration line

# Assign to the PROD symbol the task defined by
#   the icode file "producer", invoked with one argument,
#   CONS (another task).
#   producer.icn is coded to open its first argument as a
#   Stream with filemode "w".

PROD    := producer  CONS

# Activate the task assigned to the PROD symbol

@ PROD  # task_activation line

# redeclare the tasks and activate
#   the consumer first instead of the producer

CONS    := consumer  PROD -
PROD    := producer  CONS
@ CONS
```

The grammar for input lines is as follows (terminal symbols are enclosed in < > pairs, and reserved words are enclosed in double quotes):

```
input_file_line         ::= task_declaration |
                                     list_symbol_declaration |
                                     task_activation |
                                     comment

task_declaration        ::= lvalue ":=" <program_name> arguments

list_symbol_declaration ::= "list" lvalues

task_activation         ::= "@" lvalues

comment                 ::= "#" many(<&cset>)

arguments               ::= arguments argument | argument | <nothing>

argument                ::= lvalue | "$" many(<&digits>) | alphanum

lvalues                 ::= lvalues lvalue | lvalue

lvalue                  ::= alphanum

alphanum                ::= many(<&letters> ++ <&digits>)

many(c)                 ::= c many(c) | c
```

The symbols `list`, `:=`, `@`, `$`, `;`, and `#` are reserved, as are the lvalues `_HALT_`, `_LIST_`, `_NEWLINE_`, `_SCRIPT_`, `_SHELL_`, and `SHELL_MAIN`.

The following further rules apply to scripts:

- Semicolons may be substituted for line breaks.
- Names of programs that are not in the current working directory must include relative or absolute path information suitable for the platform; the program name supplied is passed directly to the `load( )` function.
- List symbols must be declared before they are used in task declarations.
- Task declarations may include forward references to other tasks.
- Task activations must be deferred until all referenced tasks have been declared.
- Once a task has terminated, the corresponding task symbol must be redeclared before it can appear in another task_activation line. Once one task symbol has been redeclared or a new task symbol has been declared, any other previously declared task symbols must be redeclared before any task can be activated. Although list symbols need not be redeclared, the lists that they represent are replaced with empty lists when new task declarations are encountered.
- If desired, string-valued arguments in task declarations may be delimited with double quotes in any way permitted by the `balq(s)` procedure (from the scan.icn file in the Icon Program Library), where `balq(s)` is invoked with a single argument. (The grammar above does not reflect the possibility of quotation.)

shell.icn - Internal Design and the `monitor_task( )` Procedure

The procedure `token( )` performs rudimentary lexical analysis, and the procedure `main( )` parses the input script. The `convert_tasks( )` procedure loads the tasks, wraps them with a monitoring co-expression (which is created by via the `monitor_task( )` procedure), and activates them.

When a co-expression has failed to produce a result, it is important that the failure be handled in a deterministic manner. For example, if task A transmits failure to task B (implicitly or using the `cofail( )` function), it is important that task B handle this failure correctly: task B must not transmit failure back to task A; otherwise, tasks A and B can go into "infinite failure", i.e., A fails to B which fails to A which fails to B *ad infinitum*.

To avoid "infinite failure" scenarios, shell.icn invokes each task via a separate co-expression that the `monitor_task( )` procedure creates. This co-expression refuses to resume the monitored task when the latter has "unrecoverably cofailed to" the former. Cofailure is "unrecoverable" when there is no co-expression that clearly should be resumed next. Thus:

- If the monitoring co-expression receives a value (or failure) from a task other than the monitored task, it pushes the reference to the transmitting task onto a list and transmits the value (or failure) received to the task that it is monitoring.
- If the monitoring co-expression receives a value (or failure) from the monitored task, and if it cannot pop a reference to a task from the list, it cofails to `&main`; otherwise, it transmits the value (or failure) to the task referenced by the value popped from the list.

There is a performance penalty for using this scheme; each task switch costs two co-expression switches instead of one.

shell.icn - Producer/Consumer and Client/Server Relationships

The principal requirements that shell.icn places on the implementation of the tasks that it runs are:

- Producers should use StreamC objects constructed with filemode "w" to communicate with (transmit data and switch control of execution to) consumer tasks.
- Clients should use StreamC objects constructed with filemode "w" to communicate with service tasks.

These rules ensure that clients and consumers always receive the first data value transmitted.

Another benefit of using StreamC to implement task switching is that, if all data values passed are (or can be) strings, many producer or consumer tasks can be tested (without running shell.icn at all) by using ordinary shells (Bourne shell, bash, csh, ksh, zsh, etc.) with standard system pipe invocation, e.g.:

```
producer - | consumer - -
```

Note well: The implementation of the tasks themselves, and how they treat their arguments, is what determines which role (producer or consumer) each task plays in a producer-consumer relationship between two tasks. It is *not* the script interpreted by shell.icn that does this. The script is merely responsible for passing to each task the appropriate arguments in the appropriate positions - what is appropriate for each task is determined by the task's definition.

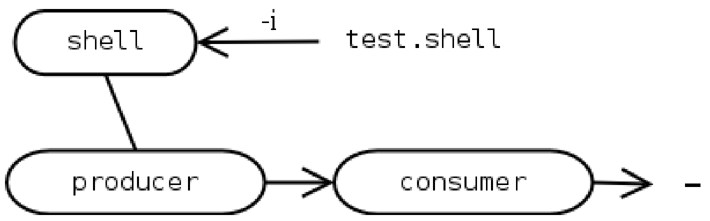
Examples of Solutions and their Components

This section presents several patterns for solutions that can be defined by input scripts for shell.icn. Regardless of each solution's simplicity, a graph of the solution is included to demonstrate the extension to the first step in the "Create Solution" use case. For brevity, the examples presented here have typically been stripped of most of their error checking and tracing code. The task programs are presented in full in the Appendix; one common difference between most of the examples here and in the Appendix is that, in the Appendix, "-t" is commonly as the first option to activate tracing of the StreamC class.

The Filter Pattern

This pattern applies tasks in a fashion very similar to how many Unix programs are applied from the command line. This pattern differs from the traditional Unix pipe-filter pattern in that the "pipelines" are only one value long.

In the test.shell example, "consumer" acts as an "output filter".



In this diagram:

- The consumer in a producer-consumer relationship appears at the head of an arrow.
- The name of an input file appears at the tail of an arrow; the name of an output file, at the head.
- Activation of a task by the shell appears as a line with no arrowheads.

The input script is as follows:

```
# test.shell
# declare tasks
CONS    := consumer PROD -
PROD    := producer CONS
@ PROD
```

It must be understood that the input script itself does <u>*not*</u> *establish which tasks are producers and which are consumers* - all that it does is to:

- Declare which symbols correspond to which tasks.
- Declare which symbols correspond to lists.
- Indicate which co-expressions should be substituted for the corresponding symbols when populating the argument lists passed to tasks as they are loaded.
- Indicate which tasks are to be activated by shell.icn directly.

The producer.icn program produces string values:

```
# producer.icn
link Stream
```

```
procedure main( argv )
  local S
  # assume that first argument is a consumer or writable filename
  S := Stream(argv[1],"w")
  S.Put( "I am the producer." )
  S.Put( "What's So Funny 'Bout Peace, Love, and Understanding?" )
  S.Close()
  # S.Close() performs cofail( S.Coexp() )
  #   for a co-expression Stream opened in "write" mode
  write("producer got resumed")
end
```

When shell.icn interprets the test.shell script, PROD simply puts two values to the output Stream (which transmits them, one at a time, to CONS) and closes the output Stream (which transmits failure to CONS).

CONS consumes each value, wraps it in quotation marks, and puts it to an output Stream. Since test.shell defines the second argument to be the string "-", the `Stream( )` constructor opens the standard output stream and instantiates the Streamf class.

```
# consumer.icn
link Stream
procedure main( argv )
  local S_in, S_out
  # assume that first argument is a producer or readable filename
  S_in := Stream(argv[1],"r")
  # assume that second argument is a consumer or writable filename
  S_out := Stream(argv[2],"w")
  S_out.Put( "I am the consumer." )
  while S_out.Put( "\"" || S_in.Get() || "\"" )
  S_out.Put( "Consumer: no more input." )
  S_in.Close()
  S_out.Close()
end
```

The output produced by "shell -i test.shell" is:

```
I am the consumer.
"I am the producer."
"What's So Funny 'Bout Peace, Love, and Understanding?"
Consumer: no more input.
producer got resumed
```

The script would produce the same output if the "task activation" line were `@CONS` instead of `@PROD`.

The resumption sequence is:

```
SHELL_MAIN (which activates PROD)
PROD (which transmits &null to CONS)
CONS (which outputs "I am the consumer" and transmits &null to PROD)
PROD (which transmits "I am the producer." to CONS)
CONS (which outputs "I am the producer" and transmits &null to PROD)
PROD (which transmits a the title of a song written by Nick Lowe to CONS)
```

```
CONS (which outputs the quote and transmits &null to PROD)
PROD (which closes the output stream, transmit failure to CONS)
CONS (which outputs "Consumer: no more input.", closes its input Stream,
      and exits, transmitting failure back to the PROD)
PROD (which outputs "producer got resumed" and exits, transmitting
      failure back to CONS)
CONS (which fails back to PROD)
PROD (which fails to SHELL_MAIN)
SHELL_MAIN
```

This sequence may be deduced by invoking shell.icn with the trace option, e.g.,

```
shell -t -i test.shell
```

Tracing the StreamC Class

To demonstrate the tracing facilities provided by Stream.icn, substitute the following code for producer.icn:

```
link Stream
global StreamC_trace
procedure main( argv )
  local S
  if
    ( type(argv[1]) == "string", argv[1] == "-t" )
  then {
    StreamC_trace := "producer.icn"
    pop( argv )
  }
  S := Stream(argv[1],"w")
  S.Put( "I am the producer." )
  S.Put( "What's So Funny 'Bout Peace, Love, and Understanding?" )
  S.Close( )
end
```

and change the task declaration for PROD to

```
PROD   := producer -t CONS
```

The output produced by "shell -i test.shell" becomes

```
 ______ (1) producer.icn: instantiating StreamC
 ______ (1) producer.icn: StreamC constructor: mode w activating p_coexp
I am the consumer.
   ... coactrace:  ______  (1)  producer.icn  is  transmitting:  "I  am  the
    producer."
"I am the producer."
 ... coactrace:  ______ (1) producer.icn received: &null
 ______ (1) producer.icn: StreamC.Put first activation produced &null
   ... coactrace:  ______  (1)  producer.icn  is  transmitting:  "What's  So
    Funny 'Bout Peace, Love, and Understanding?"
"What's So Funny 'Bout Peace, Love, and Understanding?"
 ... coactrace:  ______ (1) producer.icn received: &null
 ______ (1) producer.icn: StreamC.Put first activation produced &null
 ______ (1) producer.icn: ____ StreamC.Close cofailing stream opened for w
Consumer: no more input.
producer got resumed
```

A wide video screen may be helpful when tracing.

Passing Structures as Data Values

To demonstrate one task may transmit structures to another, substitute the following code for producer.icn:

```
link Stream
record data( one, two, three )
procedure main( argv )
  local S_next, d

  S_next := Stream( argv[1], "w" )

  d := data( "SNOBOL", "Icon", "Unicon" )

  S_next.Put( d )

  d := table( "" )
  d["first"] := "Ralph"
  d["last"] := "Griswold"

  S_next.Put( d )

end
```

and substitute the following code for consumer.icn:

```
link Stream
procedure main( argv )
  local S_prev, d

  S_prev := Stream( argv[1], "r" )
  d := S_prev.Get( )

  write( "First there was ", d.one )
  write( "Then there was ", d.two )
  write( "At last there is ", d.three )

  d := S_prev.Get( )
  write( "Many thanks to ", d["first"], " ", d["last"] )
end
```

The output becomes:

```
First there was SNOBOL
Then there was Icon
At last there is Unicon
```

```
Many thanks to Ralph Griswold
```

The Client-Service Pattern

Service Tasks

A "service task" could be characterized as "a task that behaves as an instance of a class that has only one method, which takes only one argument". This may be required when it is necessary to:

- Allow several tasks to access a single resource, such as an open file.
- Provide client tasks in several solutions with an integral unit of processing.
- Serve different numbers of clients when applying a task to different solutions.

Each time that a client resumes a service, the service yields control back to that client. (Consequently, it is not necessary to specify the client in the service's task declaration in the input script.) If the service is expected to produce a result, the service transmits the result when it resumes the client.

A client should treat a service as a consumer to ensure that the service has been activated before the client transmits values to the service. *Therefore, when a client is loaded with an argument referencing a task that is a service, it should call* `Stream( )` *with a filemode of* "*w*". A client task may use either `StreamC.Get( )` or `StreamC.Put( )` to transmit data to the service and to receive data from it; `Put( )` permits processing multiple values in a single call, whereas `Get( )` is more convenient to use when processing one value at a time.

Note well: Transmitting failure to a service task causes that task to terminate and yield control of execution back to the parent (main) task. Therefore, unless this behavior is desired, a client task should never invoke the `Close( )` *method on a service Stream.* Services must terminate upon receipt of failure to ensure that they work properly (i.e., that they will not participate in infinite failure) in a "recursive invocation" context. (See "Recursive Invocation and Argument Substitution" below for an example of recursive invocation of shell.icn.)

The `service(callback,moreargs,source)` Procedure

Stream.icn defines the procedure `service(callback,moreargs,source)` to ease implementation of services.

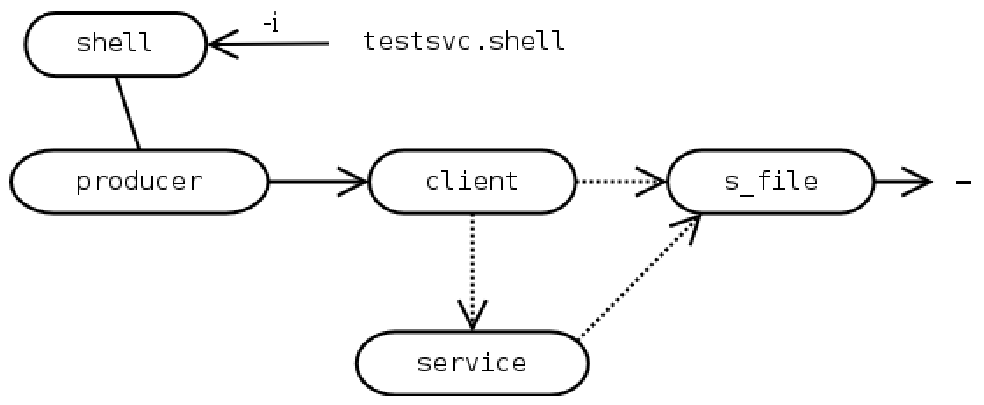
- The `source` argument defaults to the value of &source at the time that `service( )` is called; if the program implementing the service must yield to other co-expressions before invoking `service( )`, it will need to save the original &source value and pass that value as this argument if it is to behave properly.
- Once the `service( )` procedure has been invoked, the value of `source` is maintained by the procedure to reflect that &source for the last client task that activated the service task.
- The `callback` argument must be a string that names a function or must be a reference to a function. The callback function must produce a result and must take at least one argument. The first argument will be passed each result produced when `source` is resumed, and the second argument will be passed the value of the `moreargs` argument. The result returned by the callback function is transmitted back to `source` the next time that it is resumed.
- The `service( )` procedure never returns.

The heart of the implementation of the service procedure is as follows:

```
# invoke a callback function as a co-expression service
procedure service( callback, moreargs, source )
  local failed, result
  \callback | fail
  # save the &source if it was not supplied as an argument
  /source := &source

  # repeat endlessly
  repeat {

    # transmit result to source, and get next operand from source
    #   1. if source transmits failure, then abort
    if
      not (
        result := if   /failed
                  then (result @ source)\1
                  else cofail( source )\1
      )
    then
      stop( ) # terminate task and yield control of execution
              #   back to parent (main) task

    #   2. save the &source for the current resumption before
    #      invoking the callback (in case the callback needs to resume
    #      other services or co-expressions before returning)
    source := &source

    #   3. invoke callback to produce next result (or failure)
    if
      result := ( /failed, ( callback( result,  moreargs ) )\1 )
    then
      failed := &null
    else
      failed := 1
  }
end
```

Applying Service Tasks

This example (testsvc.shell) demonstrates using two programs (named service.icn and s_file.icn) as services. One service program (s_file.icn) has two clients (client.icn and service.icn) whereas the other (service.icn) has only one client (client.icn). In the following diagram of the relationships among the tasks, the service in a client-service relationship appears at the head of a dotted arrow:



The input script might read as follows. Notice that the system of tasks is declared and activated twice; in the second declaration of the SERVICE task, different arguments is passed to the service.icn program.

```
# testsvc.shell
# declare tasks
OUTPUT  := s_file   -      w
CLIENT  := client   PROD   SERVICE OUTPUT
SERVICE := service  -o     OUTPUT  my_reverse
PROD    := producer CLIENT
# activate producer
@ PROD

# declare tasks
OUTPUT  := s_file   -      w
CLIENT  := client   PROD   SERVICE OUTPUT
SERVICE := service  -o     OUTPUT  my_map   " "  "_"
PROD    := producer CLIENT
# activate producer
@ PROD
```

OUTPUT and SERVICE are both services. Notice that `@ PROD` or `@ CLIENT` may be used to activate the solution, but neither `@ SERVICE` nor `@ OUTPUT` will work.

Implementation of service.icn is much simpler given the definition of the `service( )` procedure in Stream.icn.

```
# service.icn
link Stream
global logstream     # optional stream for logging

invocable all        # do not discard any procedure names when linking;
                     #   this is necessary for string invocation

procedure main( argv )
  local source := &source # save the first source

  # enable logging if applicable (if so, the -o option must appear
  #   together with its value before the argument naming the
  #   function to be executed)
  if ( type(argv[1]) == "string", argv[1] == "-o" ) then {
     pop( argv )
     logstream := Stream( pop( argv ), "w" ) |
        stop( "service.icn: could not open log Stream" )
     }
  service( argv[1], argv[2:0] | &null, source )
end

procedure my_map( s, m )
  if (\logstream, type(m)=="list", type(m[1]) == type(m[2]) == "string" )
  then
    logstream.Put( "service: my_map("||image(s)||
                   "," || ( m[1] || "" ) || "," || ( m[2] || "" ) || ")"
                )
  /m := [ ]
  return map ! ( [ s ] ||| m )
end

procedure my_reverse( s )
  (\logstream).Put( "service: my_reverse("||image(s)||")" )
  return reverse( s )
end
```

The s_file.icn program is a general-purpose file access service. The implementation for s_file.icn is as follows:

```
# s_file.icn - service to read or write files
link Stream
global S_f # file stream created for first argument
procedure main( argv )
  local f # file produced by open( )
  local source
  source := &source
  S_f := Stream ! argv[1:0] | stop( "s_file.icn: cannot open file" )
  if find( "r", argv[2] ) then
    service( S_in_func, &null, source )
  else
    service( S_out_func, &null, source )
end

procedure S_in_func( )
  return S_f.Get( )
end

procedure S_out_func( value )
  if S_f.Put( value ) then
    return
  else
    fail
end
```

The implementation for producer.icn has been presented above. The implementation for client.icn is as follows:

```
# client.icn
link Stream
procedure main( argv )
  local S_in, S_out, S_service, data_in, data_service, data_item
  S_in := Stream(argv[1],"r")
  S_service := Stream(argv[2],"w")
  S_out := Stream(argv[3],"w")
  S_out.Put( "I am the client.")
  while ( data_in := &null, data_in :=  S_in.Get() ) do {
     if data_service := S_service.Get( \data_in ) then
      S_out.Put( data_service )
  }
  S_out.Put( "client: no more input." )
end
```

The resumption sequence interleaved with the output produced by "shell -i testsvc.shell" is

```
     SHELL_MAIN
     PROD CLIENT
       I am the client.
     SERVICE OUTPUT
       service: my_reverse("I am the producer.")
     SERVICE CLIENT OUTPUT
       .recudorp eht ma I
     CLIENT PROD CLIENT SERVICE OUTPUT
            service:  my_reverse("What's  So  Funny  'Bout  Peace,  Love,  and
Understanding?")
     SERVICE CLIENT OUTPUT
       ?gnidnatsrednU dna ,evoL ,ecaeP tuoB' ynnuF oS s'tahW
     CLIENT PROD CLIENT OUTPUT
       client: no more input.
     SHELL_MAIN
     PROD CLIENT
       I am the client.
     SERVICE OUTPUT
       service: my_map("I am the producer."," ","_")
     SERVICE CLIENT OUTPUT
       I_am_the_producer.
     CLIENT PROD CLIENT SERVICE OUTPUT
             service:  my_map("What's  So  Funny  'Bout  Peace,  Love,  and
Understanding?"," ","_")
     SERVICE CLIENT OUTPUT
       What's_So_Funny_'Bout_Peace,_Love,_and_Understanding?
     CLIENT PROD CLIENT OUTPUT
       client: no more input.
     SHELL_MAIN
```

One obvious question is, "Why not implement all consumers as services?" There are two reasons:

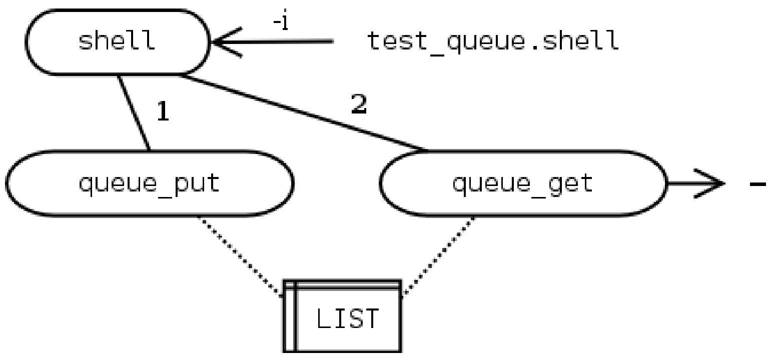
1. Service tasks that are supported by the `service( )` procedure abort when failure is transmitted to them. Consumer tasks need to be able to continue executing after they receive failure so that a producer can transmit failure to indicate that there is no more input to process.

2. Services are slightly more tricky or cumbersome to write than consumers because consumers do not need to track the first &source.
3. Consumers do not need to be written to accommodate callbacks, which can be an awkward way to express many algorithms.
4. A consumer can treat the values coming from different producers differently. A service has to respond the same way to the values that it receives regardless of the task that transmitted them.

The Queuing Pattern

Sometimes it is preferable to queue several results rather than switching tasks each time a result is generated by a producer. For example, if a producer creates both a sequence of instructions and a sequence of data, it may be preferable to queue the data required for each instruction in order to keep the consumer clear on whether a value is an operand (put onto the queue) or an operator (transmitted to the consumer when it is resumed). Alternatively, the task needing the data may not be the next consumer in the pipeline, and using a queue may be more clear or efficient than passing the data through the whole pipeline.

This example (test_queue.shell) provides a trivial demonstration of queuing using a list that is accessible by all tasks. In the following diagram of the relationships among the tasks, reference to the list is indicated by a dotted line, and numbers indicate the order in which the tasks are activated:



In the input script, the list symbol "LIST" is declared and supplied as an argument to each task that needs to access it.

```
# test_queue.shell
list LIST
PUT  := queue_put LIST
GET  := queue_get LIST -
@ PUT
@ GET
```

The main task first activates the queue_put task, which puts two items to the list and then terminates, yielding to the main task:

```
# queue_put.icn
link Stream

procedure main( argv )
  local S_list

  S_list := Stream( argv[1] )
```

```
    write( "I am queue_put.icn" )

    S_list.Put( "First string-value transmitted"
              , "Second string-value transmitted" )
  end
```

Next, the main task activates the queue_get task, which reads the values from the list and puts them to the output Stream:

```
  # queue_get.icn
  link Stream

  procedure main( argv )
    local S_list

    S_file := Stream( argv[2], "w" )
    S_list := Stream( argv[1] )
    S_file.Put( "I am queue_get.icn" )

    while ( S_file.Put( S_list.Get() ) )

    S_file.Put( "queue_get: S_list.Get has no more input" )
  end
```

The resumption sequence interleaved with the output produced by "shell -i test_queue.shell" is:

```
  SHELL_MAIN
  PUT
    I am queue_put.icn
  SHELL_MAIN
  GET
    I am queue_get.icn
    First string-value transmitted
    Second string-value transmitted
    queue_get: S_list.Get has no more input
  SHELL_MAIN
```

Recursive Invocation and Argument Substitution

Sometimes a solution may be made clearer by including "subsolutions" into its composition. This is done by writing the "supersolution" script to load shell.icn as one of the tasks. Furthermore, a subsolution script may more readily be applied to multiple supersolutions if it has "substitutable parameters" into which it may substitute arguments passed to it.

Argument Substitution

Substitutable parameters allow the user to reconfigure a input script without rewriting it. For shell.icn input scripts, a substitutable parameter is a dollar sign followed by an integer. Arguments supplied to shell.icn after the option values are substituted into the script in place of the substitutable parameters. Therefore, when the test_subst.shell script:

```
  # test_subst.shell
```

```
list LIST
PUT  := queue_put LIST
GET  := queue_get LIST $1
@ PUT
@ GET
```

is invoked with the command:

```
shell -i test_subst.shell OUT
```

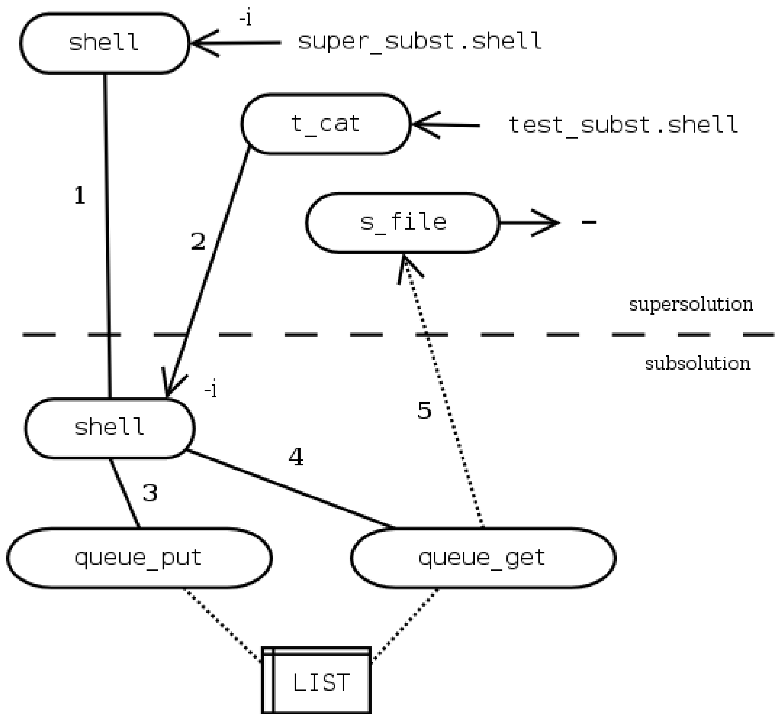
shell.icn interprets the script as if it had been:

```
# test_subst.shell, after argument substitution
list LIST
PUT  := queue_put LIST
GET  := queue_get LIST OUT
@ PUT
@ GET
```

Recursive Invocation

When a supersolution script invokes a subsolution script, shell.icn invokes another instance of shell.icn as a task. This example demonstrates:

- A supersolution invoking shell.icn with a subsolution script
- Parameter substitution in the subsolution script
- Streaming an input script to the subsolution.



The subsolution is realized by the test_subst.shell script, presented above. The supersolution is realized by the following input script, which supplies a task reference (IN) as the input-script option value, and defines the first non-option argument value to be another task reference (OUT):

```
# super_subst.shell
DATA := shell  -i                 IN   OUT
IN   := t_cat  test_subst.shell DATA
OUT  := s_file -                  w
@ DATA
```

The t_cat.icn program concatenates one or more input Streams to an output Stream:

```
# t_cat.icn
link Stream
global StreamC_trace

procedure main( argv )
  local S_in, S_out, inp, outp
  local usage
  usage := "usage: t_cat [-t] <infile>{1:N} <outfile>"

  # handle tracing option
  if ( type(argv[1]) == "string", argv[1] == "-t" )
  then { StreamC_trace := "t_cat.icn" ; pop( argv ) }

  # pull the output task or file off the end of the arg list
  outp := pull( argv )
  0 == *argv & stop( usage )
  S_out := Stream( outp, "w" ) | stop( usage )

  # send each input task or file (left to right) to output
  while inp := pop( argv ) do {
    S_in  := Stream(inp,"r") | stop( usage )
    while S_out.Put( S_in.Get( ) )
  }
  # S_out.Close( ) will cause infinite failure
  #                 if this task is the input Stream for shell.icn
  stop( ) # this is required to prevent infinite failure
          #         if this task is the input Stream for shell.icn
end
```

The reason that including `S_out.Close( )` or omitting `stop( )` would cause infinite failure when streaming input to shell.icn is under investigation; perhaps it is related to the fact that shell.icn does not use `monitor_task( )` to protect reading of the input script.

The interleaved resumption sequence and output produced by this system of solutions is:

```
supersolution.SHELL_MAIN
  subsolution.SHELL_MAIN
supersolution.IN
  subsolution.SHELL_MAIN
  subsolution.PUT
    I am queue_put.icn
  subsolution.SHELL_MAIN
```

```
    subsolution.GET
  supersolution.OUT
      I am queue_get.icn
    subsolution.GET
  supersolution.OUT
      First string-value transmitted
    subsolution.GET
  supersolution.OUT
      Second string-value transmitted
    subsolution.GET
  supersolution.OUT
      queue_get: S_list.Get has no more input
    subsolution.GET
    subsolution.SHELL_MAIN
  supersolution.SHELL_MAIN
```

In this solution, the IN task (t_cat.icn) simply copies test_subst.shell to produce an "input-script Stream" for the recursive invocation of shell.icn. Note, however, that one might make nontrivial replacements for t_cat.icn to translate arbitrary data into an input-script Stream.

The Preprocessor Pattern and a Command Line Interface Program

A "preprocessor" is a program whose response to input is to generate the input for another program. A preprocessor task could be used to generate an input script for a recursive invocation of shell.icn. Indeed, t_cat.icn in the previous example served as a trivial preprocessor, which was invoked via an additional script, super_subst.shell.

To relieve the user of the need to provide an additional script file in order to specify that shell.icn should invoke a preprocessor, the -p invocation option was implemented. The usage is:

```
shell -p <preprocessor> <optional_arguments>
```

The preprocessor must be written to accept:

- As its first argument, a task that consumes strings
- As its second argument, a list that may hold results (the "result list")
- As arguments 3 to 0, the optional preprocessor arguments

The preprocessor must produce input script lines as strings since:

- The consumer is a recursive invocation of shell.icn whose -i option value is the preprocessor task
- The `token( )` function of shell.icn relies upon string scanning functions

The script produced by the preprocessor can reference the result list using the `$1` substitutable parameter.

When invoked with the -p option, shell.icn creates an internal equivalent of an input script:

```
list   _LIST_
_SCRIPT_  := <preprocessor>   _SHELL_   _LIST_    <optional_arguments>
_SHELL_   := shell            -i        _SCRIPT_  _LIST_
@  _SHELL_
```

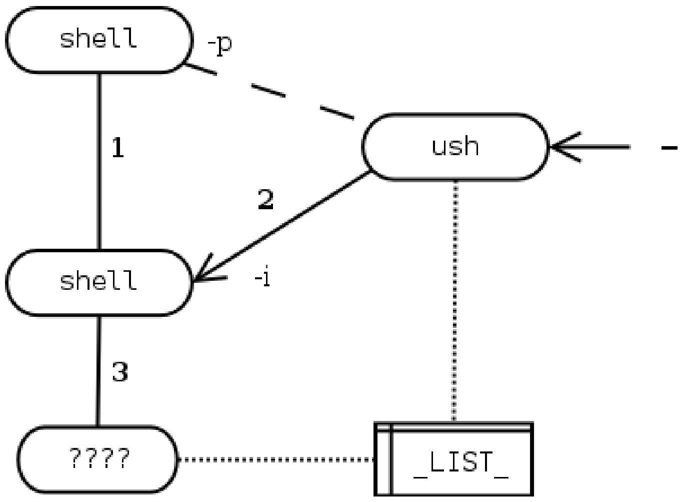
Thus, given the invocation:

```
shell -p ush -
```

shell.icn creates the equivalent of the script:

```
list   _LIST_
_SCRIPT_ := ush     _SHELL_  _LIST_    -
_SHELL_  := shell  -i        _SCRIPT_  _LIST_
@   _SHELL_
```

This script might be represented diagrammatically as:



The dashed line represents invocation of ush.icn via the internal script, and ???? represents any task or tasks that may be loaded as specified in the script which ush.icn emits and that may reference _LIST_ via the `$1` parameter.

For this example, ush.icn is a rudimentary "command-line interface" (CLI). A CLI is, in essence, a specialized preprocessor that is oriented toward having the user interactively provide input, specify where to get input, or both. The input grammar for shell.icn is not terse enough for a CLI because it favors flexibility in solution definition at the cost of needing to enter most symbols several times while defining a solution. Eventually, the ush.icn program could be expanded into a full-blown CLI program (see "Possible Future Enhancements" below), but at present it has only three functionalities:

1. If ush reads a line from the standard input that begins with the word "exit", ush terminates.
2. Else if the line read begins with a "." followed by the name of a file, that file is copied to the output Stream.
3. Else the line is copied verbatim to the output Stream.

Here is a brief interactive session, where "ush: " is the prompt for input and where values typed by the user appear in italics after the prompt:

```
shell -p ush -
ush: . test.shell
I am the consumer.
"I am the producer."
```

```
"What's So Funny 'Bout Peace, Love, and Understanding?"
Consumer: no more input.
producer got resumed
ush: PUT := queue_put $1
ush: GET := queue_get $1 -
ush: @ PUT
I am queue_put.icn
ush: @ GET
I am queue_get.icn
First string-value transmitted
Second string-value transmitted
queue_get: S_list.Get has no more input
ush: exit
```

Notice that, in this example, _LIST_ is used by the tasks in the generated script; it does not hold results for ush.icn to read, since ush.icn does not yet have code to read those results.

The essence of the ush.icn program is as follows:

```
link Stream
link scan
procedure main( argv )
  local S_in, S_out, S_list, S_src, inp, outp, result_list,
     line, ws, quoted, usage
  usage := "usage: ush <outfile> <result_list> <infile>{1:N}"
  ws := ' \t'
  argv[2] | stop(usage)
  argv[3] | ( argv[3] := "-" )

  # open the consumer of input-script strings
  outp := pop( argv )
  S_out := Stream( outp, "w" ) | stop(usage)

  # Note that ush.icn does not (yet) do anything else
  #   with the next argument once the Stream has been opened.
  result_list := pop( argv )
  S_result := Stream( result_list, "r" ) | stop(usage)

  while inp := pop( argv ) do {
    S_in  := Stream(inp,"r") | stop(usage)
    S_in.Data( ) === &input & writes( "ush: ")
    # prepend a space so the "tab(many(ws))"
    #   will always succeed at the beginning of a line
    # append " EOL" to line because "s ? balq( ws )" fails
    # if s has no spaces
    while line := S_in.Get( ) do {
      " " || line || " EOL" ?
        if ( tab(many(ws))
           , ="."
           , tab(many(ws))
           , quoted := tab( balq( ws ) )
           , quoted := ( quoted[1:2] == quoted[-1:0] == "\""
           , quoted[2:0]
           ) | quoted
```

```
          )\1 & S_src := Stream( quoted, "r" )
        then
          while S_out.Put( S_src.Get( ) )
        else if
          ( tab(many(ws)) | 1 , ="exit" )
        then {
          S_out.Put( "_HALT_" )
          stop( )
        }
        else
          S_out.Put( line )
      S_in.Data( ) === &input & writes( "ush: ")
    }
  }
  S_out.Put( "_HALT_" )
  stop( )
end
```

Performance Measurement

An informal test was run to compare the performance of the shell with a multiprocess pipe/filter solution. (Perftest_archive.sh in the second appendix can be used to extract the files needed to reproduce the test). Measurements were made on a Debian Sarge Linux installation a 650 MHz AMD K7 (Athlon) processor, 512 kb cache, and a gigabyte of memory.

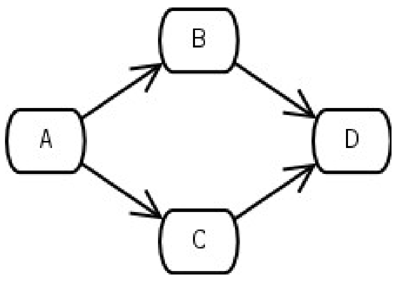
Two programs, perfprod.icn and perfcons.icn, were used to pass about 14 million characters of data from one program to another, varying the number of bytes passed per context switch. For 100,000, 10,000, and 1,000 bytes per context switch, the shell was seven-fold faster than the multiprocess solution, regardless of whether data Put to the pipe was flushed from the buffer after each Put. For 100 and 50 bytes per context switch, shell performance shrunk to about four-fold faster and two-fold faster than the multiprocess solution, respectively; for 10 bytes per context switch (and below), performance was about equal.

These results, albeit obtained under contrived conditions, seem to suggest that it is likely that composing multi-program solutions in the shell would offer at least a modest performance advantage over multi-process solutions under most real-world conditions.

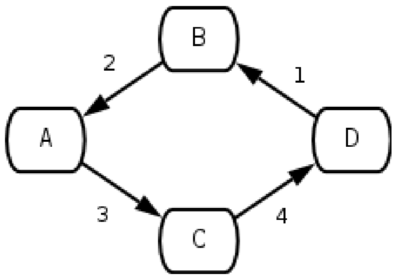
Limitations and Future Enhancements

Cyclic Relationships may have Unexpected Behavior

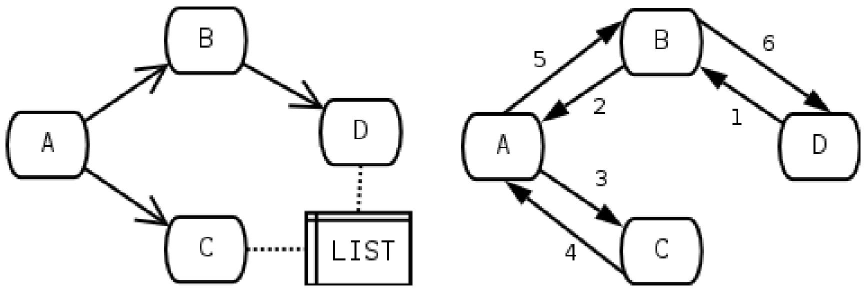
One can imagine "cyclic producer-consumer graphs" (and write corresponding input scripts); however, these might present significant problems. For example, suppose that A is a producer for B, B is a producer for D, A is a producer for C, and C is a producer for D.



If D resumes B for a result, B resumes A, A resumes C, and C resumes D, then the "resumption graph" looks as follows:



The difficulty with this is that D is defined to be a consumer of values from both B and C, yet when D resumes B for a result, it gets the result produced by C instead of a result produced by B. Unless D is a generic service that does one thing regardless of which client resumes it (such as s_file.icn), the programmer of D may expect resumption of B to produce a result that B produced instead of a result produced by C. One way around this problem is to interpose a list:



<u>Enhancements to shell.icn and Stream.icn</u>

shell.icn is a proof-of-concept implementation. Some enhancements that might make it more generally useful are:

- Employ a more sophisticated lexical analyzer and parser. The rudimentary lexical analyzer (the `token( )` procedure) passes strings to the parser, rather than passing symbol-value pairs in the manner of the lex family of analyzers; for only this reason, the strings "_NEWLINE_" and "_HALT_" cannot be used for task or list symbols.
- Modify the `token( )` procedure to analyze Streams that may produce non-text values, so that preprocessors could produce input scripts containing other values such as references to tasks and data structures.

It would probably be best to focus on adding only features that make the shell significantly more useful.

Additional Programs to Enhance Usability of the Shell

The following programs might make the shell a tool that is more generally applicable to problems that a Unicon programmer may be solving:

- Define a protocol for using the result list.
- Expand the ush.icn program to permit the user to specify a search path for icode files to load as tasks, perhaps via an environment variable, a command-line option, or both.
- Add "here document" capability so that scripts can specify standard input to the solutions that they define.
- Enhance the "." operation to permit passing of arguments to the included script.
- Permit enhanced notation to reduce or make easier the use of explicit task and list symbols. Perhaps the first three example solutions might be expressed as:

```
        producer | consumer -
        OUT := s_file - w ;
   producer | { { client ..| service -o OUT my_reverse } ..| OUT }
    { queue_put .. LIST } ; { queue_get .. LIST }
```

- Add keywords for iteration and conditional execution to control flow based on the contents of the result list.
- Create an "adapter" program that could load any existing Unicon program as a filter task for use by the shell. This would particularly valuable because it would enable programs from the Icon Programming Library to be used without modification.
- Create "adapter" programs to link with procedure definitions from the Icon Programming Library; such programs would function as service tasks so that the linked procedures would be available to other tasks.
- Create an "adapter" program that could use the `pipe( )` or `system( )` built-in functions to start a "foreign" (non-Unicon) program as a separate process. This program would allow other tasks to exchange text values with the other process.

Final Thoughts

Practical applications for a Unicon shell might be large-scale, runtime-configurable software systems. For example, imagine a cross-compiler with several target-platforms. The user could configure the output via a script so that output tasks appropriate for the chosen platform would be loaded and executed. Adding support for a new platform would simply entail writing and testing a few tasks and the scripts that invoke them. Contrast this with having to add a new invocation option to a large program for each new target program, and how a Quality Assurance Department would deduce what functionalities had been changed and would need to be tested or retested.

The shell presented here meets the requirements defined by the author and demonstrates that implementation of a shell for Unicon is both practical and relatively simple. One limitation of this work is that it only addresses the requirements of one programmer; if more programmers' insights become available, a more general set of requirements may be defined, which could in turn give rise to a more generally useful shell program.

Acknowledgments

Definition and implementation of co-expressions in Icon was a product of Steve Wampler's Ph.D. research under Ralph Griswold. The same is true for definition and implementation of the transmission of values between co-expressions. MT Icon was a product of Clint Jeffery's Ph.D. research with Ralph Griswold.

Steve raised a question that suggested the Client-Service pattern demonstrated in testsvc.shell.

Both Clint and Steve provided rapid (and at times crucial) assistance during the research and development of the code for Stream.icn and shell.icn; both provided critical review for this report.

The `Stream( )` class constructor takes some inspiration from an Icon implementation exercise that proposed creating a `stream` Icon datatype [10]. Although, in principle, the Stream class reported here has some similarities with the suggested `stream` datatype and was intended to solve some of the same problems, it differs in the operations implemented and avoids changing the implementations of the language and the virtual machine.

References

1. http://en.wikipedia.org/wiki/Unix_philosophy
2. http://en.wikipedia.org/wiki/Pipes_and_filters
3. http://en.wikipedia.org/wiki/Interprocess_communication
4. Knuth, Donald, 1968. *The Art of Computer Programming, Volume 1.* Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., Section 1.4.2.

5. Wampler, Steven B., and Griswold, Ralph E., 1986. *Co-Expressions in Icon.* Computer Journal 26: 72-78.
6. Griswold, Ralph E. and Griswold, Madge T., 1996. *The Icon Programming Language, 3rd edition.* San Jose, California: Peer-to-Peer Communications, pp. 118-121.

7. Jeffery, Clinton L., 1997. *The MT Icon Interpreter.* (Icon Project Document 169), http://www.cs.arizona.edu/icon/docs/ipd169.htm
8. Clinton Jeffery, Shamim Mohamed, Robert Parlett, and Ray Pereda, manuscript in preparation. *Programming with Unicon.*

9. http://unicon.sourceforge.net/
10. Griswold, Ralph E., and Griswold, Madge T., 1986. *The Implementation of the Icon Programming Language.* Princeton, New Jersey: Princeton University Press, p. 310.

Appendices

Please view the first appendix on-line at http://unicon.org/generator/v2/shell_appendix_sh.html.

The second appendix is available at http://unicon.org/generator/v2/perftest_archive.sh.

Fun with Co-expressions, part three: Co-expressions as Closures

Steve Wampler

A *closure* is a representation of a function along with the lexical environment in which the function was created[3]. More importantly, a closure represents a function that can be created dynamically. Furthermore, the definition of this function may reference variables in the surrounding lexical context. Closures can be found in a large number of high-level languages and is a particularly valuable tool found in many functional programming languages. As an example, the following pseudocode is a solution to a programming challenge [4]:

```
Function genAcc(n) {
   return newFunction(i) {
       return n := n + i
       }
   }
```

Function `genAcc` returns a dynamically created function that computes the sum of the value of its argument `i` along with the value passed originally to `genAcc` as `n`. It also increments the value of `n`. Continuing this example:

```
x := genAcc(3)
y := genAcc(5)
write(x(4),” “,y(4))
write(x(2),” “,y(3))
```

would output:

```
7 9
9 12
```

Another way to view a closure is in terms of binding-times. The above closure can be thought of as defining a new function of two parameters `n` and `i` where the value of `n` is bound at the time the function is defined while the value of `i` is bound when the function is invoked. In the above pseudocode, the parameter `n` is *implicitly* available in the closure. It is also possible to define a closure mechanism in which the early-binding of function parameters is *explicitly* specified.

Co-expressions as closures

It turns out that co-expressions are a natural implementation of closures. By definition, a co-expression has implicit access to the variables found in the surrounding lexical context. A first step, then is to translate the above pseudocode into a near Unicon form that binds the value of `n` as the co-expression is defined.

```
procedure genAcc(n)
   C := create co-expression referencing n
   return C
end
```

Each call to `genAcc()` produces a co-expression where the value of `n` is bound to the argument to `genAcc`. This provides the early binding needed on `n`.

All that needs to be done now is to write the co-expression so each activation behaves as one would expect a function call to behave. That is, the co-expression must accept input through a parameter and produce the desired accumulation on `n`.

The solution is to take advantage of the explicit messaging capability that is available when viewing co-expressions as coroutines. There are several key aspects using co-expressions as coroutines that play a role here:

- *All* expression evaluation in Unicon (and Icon) can be thought of as being performed inside a co-expression. In particular, it's perfectly reasonable to think of the evaluation of a Unicon program as initiating from:

  ```
  @(create main(args))
  ```

- Every co-expression can explicitly activate any other co-expression, *including the one that activated it*. The keyword `&source` provides convenient access from the current co-expression to its activator.

- You can pass a value from one co-expression to another through explicit activation. So `outValue@&source` passes the value of `outValue` to the source co-expression.

- Since evaluation of a co-expression pauses at the point of explicit activation and resumed from that point, any value passed through explicit activation of the paused co-expression becomes the result of that paused evaluation.

The last two points can be confusing but, when combined, mean that the expression:

```
inVal := outVal@&source
```

does the following:

- Whenever `outVal@&source` is evaluated, the value of `outVal` is passed to the activating co-expression.

- Execution pauses in this co-expression while the activating co-expression is evaluated.

- When execution resumes in this co-expression, any value explicitly passed to it is promptly assigned to `inVal`.

This behavior is exactly what is needed to implement the target closure. The missing co-expression can be written as:

```
C := create while i := n@&source do n +:= 1
```

Only one problem remains. The co-expression C, when created, is paused at the start of the expression. This isn't what is desired here, as any value passed in on activation will be discarded. Instead, the co-expression needs to be *pre-evaluated* so it is paused waiting for a new value to assign to i. An initial activation of C will evaluate C to the activation of &source, leaving the co-expression paused at the desired point. (This first activation of &source receives the initial value of n from C, but that value can simply be ignored.)

Interestingly, this initial activation has the effect of rotating the three bullet points above to read:

- Execution pauses in this co-expression while the activating co-expression is evaluated.
- When execution resumes in this co-expression, any value explicitly passed to it is promptly assigned to `i`.
- Whenever `n@&source` is evaluated, the value of `n` is passed to the activating co-expression.

So the final Unicon/Icon form for producing the target closure becomes:

```
procedure genAcc(n)
   C := create while i := n@&source do n +:= 1
   @C    # position C to accept value for i
   return C
end
```

(the three executable lines could be combined into a single line at the expense of clarity, of course.) A simple main procedure to test the above code is:

```
procedure main()
   x := genAcc(3)
   y := genAcc(5)
   write(4@x," ",4@y)
   write(2@x," ",3@y)
end
```

which produces the same output as the original pseudocode.

Strictly speaking, this Unicon code is only a partial solution to the actual programming challenge. The challenge also requires that the invocation of the generated operation be syntactically identical to 'normal' function invocation. The syntax for activating a co-expression does not meet that requirement.

A template for building closures

The above example provides a foundation from which it is possible to derive a generic template that can be used to build arbitrary closures. In near-Unicon, this template may be written as:

```
procedure mkClosure(eb_params)
   local C, lb_param
   C := create while lb_param := result@&source do {
            Actions computing result from eb_params
               and lb_param
            }
   @C   # Position C to accept lb_param
   return C
end
```

Here, `eb_params` is the set of closure parameters that are to bound to values when the closure is defined, and `lb_param` holds the (single) parameter bound when the closure is invoked. The previous example follows this template but uses the variable n as both the early bound parameter and as the result.

This template works fine for closures where there is a single late-bound parameter, but what about closures requiring more than one such parameter? Only a single value can be passed to a co-expression using explicit activation but this value can, of course, be a structured value. So for example, to invoke a closure with the three late-bound parameters 3, 4, and 5, one could write:

```
[3,4,5]@C
```

and use list subscripting inside the closure action to reference the individual parameters.

This use of a list to pass values into a co-expression presents an unappealing syntax. It would be nice if co-expression activation had an alternative syntax as:

```
C(input_params)
```

(This syntax would also permit a fully-compliant solution to the above programming challenge!). In its simplest form, this syntax could be automatically translated into `[input_params]@C`. Other, more sophisticated implementations are also possible, but would require additional support for accessing input values within a co-expression.

There are, of course, tradeoffs to consider. While the above proposed syntax does provide a clean means of passing multiple values into a co-expression, it also hides the fact that a co-expression is involved in the control flow at this point. (Some people may consider that a positive change.) If the proposed operator overloading implementation is extended to support the procedure call operation then perhaps that might provide a convenient means of experimenting with this alternative syntax prior to committing to integrating it into the language. Also, this syntactic alternative adds pressure for adding support for more conveniently handling multiple input values *within* a co-expression definition, increasing the complexity of the language.

All of the examples shown here have encapsulated the co-expression creation in a separate procedure. While this helps clarify the closure's definition by constraining and identifying the early-bound parameters, there is no technical requirement imposing this encapsulation. For example, the following code is equivalent to the previous example:

```
procedure main()
   local n
   n := 3
   @(x := create while i := n@&source do n +:= i)
   n := 5
   @(y := create while i := n@&source do n +:= i)
   write(4@x,” “,4@y)
   write(2@x,” “,3@y)
end
```

This works because every co-expression captures a snapshot of the environment in which it is created, so each co-expression in the above code retains its own copies of n and i.

Other approaches

Co-expressions are not the only way to express closures in Unicon. The Unilib[5] library includes a `Closure` class that provides an alternative approach. For example, using `Closure`, the previous pseudocode example can be written as:

```
procedure genAcc(n)
   return Closure(accum, [n])
end

procedure accum(n, i)
    return n[1] +:= i
end
```

An example main program using the above is:

```
procedure main(args)
   x := genAcc(3)
   y := genAcc(5)
   write(x.call(4),” “,y.call(4))
   write(x.call(2),” “,y.call(3))
end
```

this alternative technique for generating Closures will be covered in depth in another article.

References

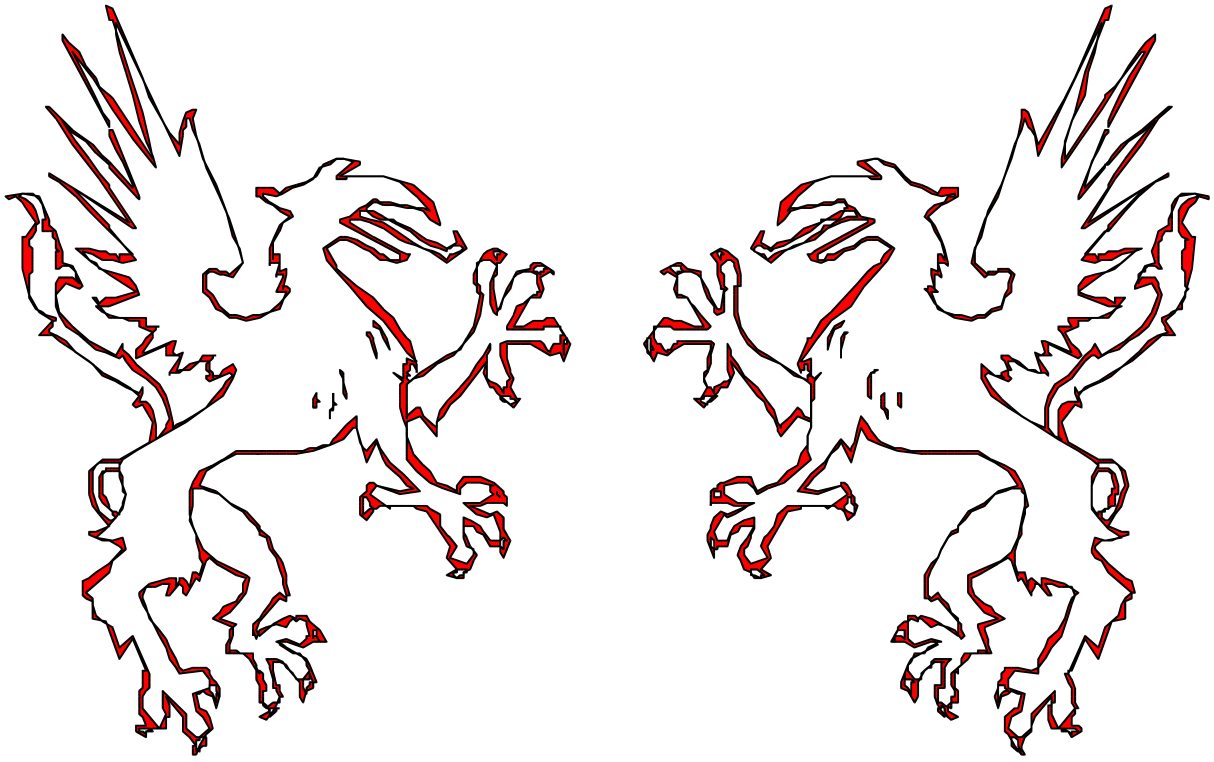
1. Wampler, Steve, "Fun with co-expressions, part one", *The Generator*, Vol. 1, No. 1, 2004, pp 8-9.

2. Wampler, Steve "Fun with co-expressions, part two", *The Generator*, Vol. 2, No. 1, 2006, pp 16-24.

3. See http://en.wikipedia.org/wiki/Closure_%28computer_science%29.

4. This example appears at http://www.paulgraham.com/icad.html, with sample solutions in a variety of languages at http://www.paulgrahm.com/accgen.html.

5. See http://tapestry.tucson.az.us/unicon.



Interoperating With PHP

PHP stands for Perl Hypertext Processor. It is a popular way of introducing dynamic server-side behavior into a webpage, easier than using CGI or a servlet engine. PHP programs mostly look like HTML documents with some dynamic elements. Of course, PHP can't do everything that Icon and Unicon can do, and there are times when it is desirable to use both.

There are two ways that PHP and Unicon can interoperate: a PHP program can invoke an external program (such as an Icon or Unicon program), or a Unicon program can write out PHP and invoke the php translator as a postprocessor. Invoking Unicon from PHP looks like

```
passthru("/your-path/your-unicon-program");
```

This is a PHP statement. While the Unicon program should write out HTML, it is not a CGI and should not expect a CGI environment. In order to use PHP from within a Unicon CGI script, you will need to have a PHP implementation which supports a `php` standalone executable, not just a PHP that runs built-in to your web server. For example, your machine might have a `/usr/bin/php`. In that case, using PHP in your CGI is simple, just open a pipe for writing, and write your output to the pipe.

Underdocumented Unicon

Much of what is underdocumented is that way for a reason: just because it is in CVS doesn't mean its design is finished or that its implementation is bug-free. For example, Unicon's CVS has substantial Voice Over IP and Pattern Matching facilities, but they are underdocumented on purpose, since they are likely to change.

One feature in Unicon that probably isn't in Icon is this: key(r) generates the field names of record r. There is already a function fieldnames() for this, the reason to extend key() is to increase the polymorphism between records and tables, so that heterogeneous data structures may contain either (or a mixture of both). Another recent tidbit is that pull(L, n) pulls/removes the last n elements from a list L. This is liable to be much faster than executing a loop that calls pull() many times.

The final feature for this issue is in the graphics subsystem. Traditional Icon graphics facilities report keyboard keys as strings, and mouse presses, drags, and releases as small negative integers. This input model was optimized for simple applications in the late 1980's where minimizing the network consumption (under the X Window System) was a major criterion. It works for a majority of GUI applications, but some programs need lower level input control, notably video games.

Unicon includes an attribute named inputmask that allows an application to request additional types of window input events. The attribute takes string values. The letter "k" requests keyboard release events (in addition to the press events that are reported normally). The letter "m" requests mouse motion events, even when no mouse button is being dragged. The letter "c" requests window closure events to be reported.