A Stream Data Type for Icon*

Kelvin Nilsen

TR 88-10

ABSTRACT

The string scanning environment of Icon [1] consists of two global variables: &subject and &pos which represent the string argument of scanning and the current scanning position within that string respectively. In general, a scanning expression is comprised of several subgoals, each of which must be satisfied in order for the scanning expression to succeed. Whenever a particular subgoal cannot be satisfied, backtracking to the most recently satisfied goal occurs automatically, and an attempt is made to satisfy that goal in some alternative way. As subgoals fail, backtracking of control causes the position component of the scanning environment to be automatically restored to its value before the failed subgoal began to execute.

This paper describes a new abstraction, called a stream, that combines &pos and &subject into a single object. Streams represent not only string data, but also data stored in files or arriving asynchronously from a user's keyboard. This allows, for example, pattern matching to be performed directly on the contents of a file, without the necessity for an explicit read operation. Further, lists can be converted into streams, permitting the same control structures that have proven useful in string scanning to be applied to more complicated problems such as list scanning. An experimental version of Icon, called Conicon, has been implemented in which streams replace files, and stream scanning replaces string scanning. This paper discusses the motivation and design of the stream data type.

February 12, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

A Stream Data Type for Icon

Introduction

One of the principal application areas served by Icon is the analysis of string data. However, there is a variety of problems related to the processing of string data for which the current facilities of Icon are not well suited. Many of these applications come from the special-purpose domain of real-time communications programming. Others, however, are batch applications for which changes in Icon's string scanning facilities might benefit large numbers of users.

For example, because of the structure of Icon's input facilities, Icon programs that perform string scanning on data from a file typically iterate, scanning one line of the file on each pass through the loop. However, many real-world patterns span multiple lines from the input file. Suppose that a user wants to build a scanning expression to find a C-style comment in a data file. In C, comments begin with the string "/*" and end with "*/", between which any characters, including newlines, may appear. Though it is possible to write a scanning expression to recognize this pattern by combining scanning procedures with read invocations, the solution is awkward. Patterns that match multiple lines of a data file are difficult to program because the individual lines arrive from the data file one at a time. This is a problem not only because it is not possible to anticipate how many lines may be matched by a particular pattern, but also because string scanning conventions require that a scanning expression be able to backtrack to the most recently satisfied subgoal if subsequent subgoals cannot be satisfied. This might require, for example, that certain lines from the data file be unread. One solution to the problems discussed here is to read the entire data file into memory before attempting to scan. Of course, this solution requires enough memory to store the data file, and could impose unnatural timing delays on asynchronous file data such as might arrive from a UNIX[1] pipe or user's keyboard.

Another application for which Icon's scanning capabilities are not particularly well suited is goal-directed parsing of any data structure other than a string. For example, a lexical analyzer might produce a list of tokens. A natural approach for implementation of a parser would be to scan the list items using the established conventions of string scanning. Because of the similarities that exist between strings and lists, it is possible to apply the conventions of string scanning to the slightly more complicated problem of list scanning. This capability, not readily available in Icon, is a natural extension of the stream data type provided by Conicon.

The discussion above presents two traditional scanning problems that might benefit from an alternative approach to string scanning. The stream data type was originally motivated, however, by a desire to apply Icon's high-level string processing ideas to communications applications. Programs that communicate with external systems in general transmit and receive streams of information, which is most commonly formatted as packets of characters. In order to understand the received information, the recipient of a stream of characters must determine its intended structure. This consists, for example, of recognizing the beginning and end of a data packet and assigning some meaning to the packet's contents.

Scanning of streams of characters, as might arrive from some remote computer, is difficult for many of the same reasons that were mentioned above. Newline characters may appear at arbitrary points within a data packet, or may not appear at all in a sequence of several packets. It is not acceptable in most communications protocols to delay processing of a block of text until a newline character arrives. Ideally, when scanning potentially infinite streams of characters, the scanning expression would be allowed to view as much of the input stream as has arrived from the remote system, and failure of subgoals within the scanning expression would return data to the stream's input queue so it can be processed by further scanning attempts. Because communications software must respond quickly to the data that is sent from external systems, language capabilities for accessing each byte of stream data must execute in constant time. It is also important to restrict the amount of information required to represent a stream in order to implement the abstraction on a machine with limited memory.

---

[1] UNIX is a trademark of AT&T Bell Laboratories.

1. The Stream Data Type

Conicon, a dialect of Icon with language features designed to facilitate communications programming, provides a stream data type in place of the file type. In Conicon, &input, &output, and &errout are all streams. Scanning in Conicon has been modified to operate on streams instead of strings and &subject initially represents &input. Conicon has no &pos keyword.

The open function in Conicon returns a stream either for reading or writing, depending on the mode supplied as its second argument. As in Icon, open's first argument is a string representing the name of a file. The second argument is a string representing the mode with which the stream should be opened. If no mode is specified, the default of "r" is supplied. The available modes for opening system files are:

r	open for reading
w	open for writing
a	open for writing in append mode
p	open a pipe to a process

The 'p' option is used to start up a system process for which either its standard input or standard output is represented by the stream returned by open. When combined with the 'p' mode, the 'r' option specifies that the stream, which is opened for reading, represents the standard output of the process. If instead, the 'p' option is combined with a 'w', the stream representing the standard input of the process is opened for writing. So, for example, the following lines send a mail message to user kwalker:

```
msg := open("mail kwalker", "pw")
write(msg, "hi ken\n")
close(msg)
```

If neither the 'r' nor 'w' options accompany a 'p', the stream is opened for reading and represents the output of the process.

Streams are also created by coercion of other objects. This coercion occurs, for example, if a string or list is supplied as the argument to a scanning expression. Any data that can be coerced to a string can be coerced to a stream representing that string. Explicit conversion to streams is provided by the stream function. stream fails if it is unable to convert its argument to a stream.

Writing to a stream opened for output behaves similarly to Icon. If the first argument to write is a stream, all subsequent arguments are written to that stream provided it is opened for writing. If the stream is not opened for writing, Conicon aborts with a runtime error message. If write's first argument is not a stream, then all arguments are written to &output. write returns the number of characters written. Unlike Icon, no newline is automatically written to the stream after the last argument. Because of this, there is no need for a writes function in Conicon. Also, Conicon's version of write does not allow users to change streams within the argument list by supplying another stream argument.

Reading from streams opened for input is accomplished by two functions: probe and advance. probe allows the user to view the contents of a stream without advancing the current position within that stream. Positions within a stream are numbered relative to the current position in the same way the positions are numbered within strings and lists. This means that, immediately after converting a string or list to a stream, the positions within the stream are numbered exactly as they had been numbered in the string or list from which the stream was derived. The second argument to probe, which defaults to &subject, specifies the stream on which probe operates. For streams of characters, probe returns a string representing the data between the stream's current focus and the position named by its first argument. For example, the following expression assigns "fee" to the variable s:

```
"fee fi fo fum" ? (s := probe(4))
```

The second argument to probe, which defaults to &subject, represents the stream to be read by probe. For streams of objects (as might be created from a list), probe returns a list representing the data between the current focus within the stream and the position specified by its first argument. The probe invocation below, for example, returns a two-element list containing the strings "while" and "(".

```
probe(3, ["while", "(", index, "+", "4", ">", "12", ")"])
```

With either type of stream, if a third argument is supplied to probe, it represents an initial offset to which the

current focus for the stream is temporarily advanced before determining the absolute position named by probe's first argument. The following code assigns "fum" to s.

```
"fee fi fo fum" ? (s := probe(4,,-3))
```

probe fails if its stream argument does not have enough remaining items to produce the requested string or list.

advance expects the same arguments as probe and produces the same result, but advance has the side effect of advancing the current position for the stream to the point named by its first argument. Notice that Conicon has no read function. It is a simple matter to build read[2] out of advance and upto, which behaves similarly to the upto function of standard Icon:

```
procedure read(s)
  /s := &input
  return s ? 1(advance(upto('\n')), advance(2))
end
```

If advance is resumed, it restores the stream's focus to its previous value. Suppose that &subject represents a stream of tokens, each token stored as a string. The code fragment below might be used to parse the header of a C while statement:

```
if advance(2)[1] == "while" then {
  advance(2)[1] == "(" | stop("expecting left parenthesis")
  (parse_expr() & advance(2)[1] == ")") | stop("expecting right parenthesis")
  }
```

In the code fragment above, advance(2) advances the stream to position 2 relative to its current focus and returns a list of the data that is found between the stream's old and new points of focus. Note that the returned list is of length one. If the single entry in the list represents the while token, the body of the the controlling if statement is executed with the stream focused on the next token. On the other hand, if advance fails, the body of the if statement is not executed. It is also possible for advance to succeed but for the comparison to fail. In that case, advance is automatically resumed and the stream's focus is restored to its previous value.

Following the while token, only a left parenthesis may appear. Assuming that tokens representing the while keyword and a left parenthesis are matched, an attempt is made to parse an expression. This goal-directed parser simply allows parse_expr to generate from shortest to longest each point at which a valid subexpression has been parsed. For example, in parsing the expression:

```
index + 4 > 12
```

parse_expr treats index, index + 4, and index + 4 > 12 each as valid subexpressions. If this expression is supplied as the controlling expression of a while statement, parse_expr would suspend three times. Since neither of the first two suspensions is followed by a right parenthesis, parse_expr would be resumed in order to find alternative ways to satisfy its goal[3]. Most programming languages are designed such that parsing them does not require all of the generality of a backtracking parser. However, certain real problems require more powerful techniques than are available from standard parsing algorithms. For example, the expressiveness of Conicon for the implementation of goal-directed parsers might be useful in experimenting with algorithms for the real-time parsing of natural language voice input.

Since, in Icon, the controlling subexpression of an if expression is bounded, it is not possible to backtrack into that subexpression once the body of the if expression has been entered. Thus, it is not possible for the stream's focus ever to return to the while token that was produced by advance. Conicon [2] recognizes this situation and reclaims stream data that is no longer accessible to the program.

This is an important characteristic of the stream data type. It limits the amount of information that must be retained for each stream. This allows scanning of infinitely long streams of data, that might be sent, for example,

---

[2] This read procedure differs slightly from the standard read function provided in Icon. Unlike the library function, this read requires that a newline terminate the file.

[3] This example demonstrates some of Conicon's capabilities. It is not intended to suggest that this is the most appropriate implementation of a parser for the C language.

from a base station to a weather satellite, as long as the matching expressions themselves perform only a limited amount of backtracking. This characteristic is used in the following code to search for a C-style comment:

```
&input ? {
  while advance(2) ~== "/" | {advance(2); probe(2) ~== "*"}
  if advance(2) then {
    while advance(2) ~== "*" | {advance(2); probe(2) ~== "/"}
    if advance(2) then
      write("found comment\n")
    }
  }
```

This could be rewritten to use scanning procedures:

```
&input ? {
  if advance(upto('/')) & advance(match("/*")) then
    if advance(upto('*')) & advance(match("*/")) then
      write("found comment\n")
  }
```

In contrast with the previous solution, this approach keeps more stream history around in case backtracking is required. For example, while looking for the start of a comment, all string data from the the start of &input to its current focus is retained for possible backtracking. Although the second solution is much cleaner conceptually, it is less efficient in terms of memory usage if long segments of the input file contain no comments. One approach that might be taken to give the second solution the same efficiency as the first is to revise the definitions of the scanning functions.

This is the approach taken in the following example. Here, a procedure searches for valid data packets and returns each data packet that is found, treating any data that does not fit the definition of a data packet as noise and ignoring it. This activity, carried out by many communications protocols, is analagous to lexical analysis in a compiler. Each data packet is prefaced by a special start symbol, represented below by SOH. The data packet itself is comprised of header and data components, each of which is protected by a CRC checksum. The header has a fixed length of eight bytes, but the size of the data component is specified by the first three bytes of the header.

```
record packet(hdr, data)

procedure next_packet()
  local hdr, data

  return &input ? {
    while skipto(SOH) & chk_crc(hdr := advance(10)[2:0]) do
      if chk_crc(data := advance(hdr[1:4]+1)) then
        break packet(hdr, data)
    }
end
```

In the example above, chk_crc performs an internal consistency check on its string argument, succeeding only if the string's CRC checksum has a particular value. skipto is similar to upto, but it automatically advances the stream focus as it goes. An implementation of skipto is provided below:

```
procedure skipto(c, s)
  local char

  c := cset(c)
  /s := &subject
```

```
    repeat {
      char := probe(2, s) | fail          # fail if stream is exhausted
      if any(c, char) then
        suspend 1
      advance(2, s)
      }
end
```

next_packet is written in high-level Conicon, yet executes in real time and uses only a small amount of memory to represent the incoming stream of data. Real-time performance is guaranteed by the fact that each block of code that is executed consumes at least one byte from the incoming stream. skipto consumes one character each time it executes its loop. Once skipto finds a character in c, it suspends, allowing the next subgoal to execute. The next subgoal attempts to verify that the eight bytes following the SOH character represent a packet header. If the CRC check, which is performed in time proportional to the length of its string argument, succeeds, the body of the while statement is entered. If the CRC check fails, control backtracks into the skipto procedure, returning the stream's focus to the SOH character. skipto advances past this character and searches for another SOH. Even this error condition is handled in real time. The cost, in this case, of consuming the SOH character is the cost of executing one iteration of skipto's loop and failing to verify that the following eight bytes have a correct CRC checksum. This continues until a valid packet header is found. Upon entry into the while statement's body, the backtrack point left by advance when the argument to chk_crc was computed is automatically discarded. Inside the while statement's body, a second CRC check is performed. If this check fails, next_packet assumes that data was corrupted at some point following the most recently verified packet header. The while expression loops, looking for a new SOH starting with the character following the preceding packet header. In terms of the real-time cost analysis, the time spent failing to match the data component of a packet after successfully matching a header is charged to the nine characters that were consumed by the packet's preface and header. If, however, the CRC check of the data component succeeds, the complete data packet is returned to the calling environment. Since chk_crc executes in time proportional to the length of its string argument, this execution path also satisfies real-time constraints.

To facilitate experimentation with modified versions of the string scanning procedures as was demonstrated above with the skipto procedure, these procedures are currently implemented in Conicon. Below, for example, is an implementation of find:

```
procedure find(s1, s2)
  local str, len, i

  s1 := string(s1) | stop("bad argument to find")
  len := *s1

  /s2 := &subject

  every i := seq(1) do {
    str := probe(len+1, s2, i) | fail          # fail if the stream is exhausted
    (s1 == str) & suspend i
    }
end
```

Icon's unary = operator is not available in Conicon.

Note that the parameters to find shown above are slightly different than for Icon's function by the same name. This find takes only a string argument representing the string for which to search, and a stream in which to search for it. As implemented in Conicon, the parameters of other scanning procedures have been modified similarly.

Certain aspects of lexical analysis are somewhat awkward to implement in traditional Icon. This is because the string scanning environment that might be established inside of the procedure that looks for tokens is lost each time the procedure returns. But with stream scanning, this is not a problem. When the lexical analyzer advances its focus within its input stream, this change is stored as part of the stream's internal state. Subsequent attempts to view data from that same stream automatically start where the last advance left off. Below, for example, is code for a simple lexical analyzer that reads from standard input:

```
procedure get_token()
  static firstchars, idchar, digit

  initial {
    digit := '0123456789'
    idchar :=  &lcase ++ &ucase ++ '_' ++ digit
    firstchars := idchar ++ digit ++ '*/-+()'
    }

  return &input ? {
    skipto(firstchars) &                # skip nonsense characters
    if any(digit) then                  # scan an integer
      advance(many(digit))
    else if any(idchar) then            # scan an identifier
      advance(many(idchar))
    else                                # scan an operator
      advance(2)
    }
end
```

Streams integrate naturally with Icon's goal-directed expression evaluation. For example, the scanning expression above could be rewritten as:

```
&input ? (skipto(firstchars), advance(many(digit | idchar) | 2))
```

The examples above emphasize some of the differences between string and stream scanning. There are many other applications for streams, however, that take advantage of the similarities between stream and string scanning. Below is a procedure that counts the number of times each word occurs in an input file. This solution, which uses stream scanning, strongly resembles a string scanning solution to the same problem. In the traditional solution, a loop that extracts words from a line of input is nested within another loop that reads each line from the input file. The higher level view of data files afforded by the stream abstraction eliminates the need for nested iteration:

```
procedure main()

  wchar := &lcase ++ &ucase ++ '\'-_'
  words := table(0)

  while skipto(chars) do words[advance(many(chars))] +:= 1

  wlist := sort(words, 1)
  every pair := !wlist do
    write(left(pair[1], 15), right(pair[2], 3), "\n")
end
```

Note that this program takes advantage of the initial scanning environment being automatically established with &subject equal to &input.

Though Conicon is capable of reclaiming much of the memory used to represent a stream that is no longer being accessed, Conicon does not communicate to the operating system that the file associated with the stream is no longer needed. The close function serves this purpose. After a stream has been closed, further attempts to write to the stream are treated as fatal run-time errors. Any attempt to read beyond the end of the data available at the moment the stream was closed results in failure. probe and advance requests succeed if they access only data that had already been read from the operating system at the time the stream was closed. For example, after closing a stream with the following expression, a minimum of 80 additional characters can be read from the stream s.

```
advance(81, s) & close(s) & &fail       # read-ahead 80 and close, then backtrack
```

2. Summary and Conclusions

The stream data type provides the essential capabilities of Icon's file type, encapsulates a string scanning environment into a single object, and extends scanning capabilities to include the scanning of lists in addition to strings. In many batch-oriented situations, the stream data type is easier for programmers to use than Icon's existing facilities. Furthermore, the special requirements of real-time communications programming are better served by the stream data type than by the traditional I/O facilities of Icon.

3. Acknowledgements

Although some of the characteristics of the stream data type were originally conceived as part of the design of CommSpeak [3], probing questions asked by Ralph Griswold motivated an attempt to integrate the stream data type with Icon's notion of string scanning. The design of the stream data type has benefitted from discussion with members of the Icon research group at the University of Arizona. Other members of this group are Dave Gudeman, Janalee O'Bagy, and Ken Walker. In particular, this group sat through several presentations describing my thoughts on the stream data type and provided helpful criticism and suggestions for improvements. Ralph Griswold has read several drafts of this document, and suggested a variety of improvements.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. K. Nilsen, "Garbage Collection of Strings and Linked Data Structures in Real Time", *Software—Practice & Experience*, To Appear.
3. K. Nilsen, "The CommSpeak Language Reference manual", The Univ. of Arizona Tech. Rep. 87-4, Tucson, Arizona, 1987.