The LISP Destructive Stream Facility.

In Stoy and Strachey[2] *streams* were proposed as vehicles for the transfer of information in systems. In this system we copy their concept to a large extent.

Burge[1] has posited an even more abstract and general stream model in which *destructive streams* are a special case. We can recognize the Stoy and Strachey model as this most interesting special case. A *destructive stream* is a stream which has private storage within itself which undergoes updating. This allows successive items of a stream to occupy the same storage. In Burge's more general model ordinary streams are applicable functions, are not 'destructive', are retainable and capable of backup. In his model "A *stream* is a functional analog of a coroutine [3, 4] and may be considered to be a particular method of representing a list in which the creation of each list element is delayed until it is actually needed.".

In LISP these abstract streams are defineable, but we choose to suggest a data-structure model for streams and basic stream-data-structure manipulation facilities. We choose to use a data-structure, instead of using functionals, for reasons of efficiency and to allow updateability. The stream data structure is simply a pair, the first element of which is the current item at the head of the sequence, the second element of which serves to define the rest of the sequence. Streams are described below syntactically:

A *stream* is (*heads* • *rests*) where,
heads is the next item of the stream which can be any *s-exp*, and
rests serves to define the rest of the stream and is either:
a *stream*, or
a special stream description, *stream-desc*, *<rfn bd asc [any...]>*, where:
and *rfn* the stream dependant function is an *e*,
bd the fast-buffer description is NIL in the case of *slow-streams*
or *< string begindex curindex endindex>* for *fast-streams*,
where
begindex the beginning index is a {0 | 1 | ...}, and
curindex the current character index is a {0 | 1 | ...}, and
endindex the boundary index is a {0 | 1 | ...}, and
and *asc* the *associated-states* which is an *a-list*,
and *any* is any stream dependent information that the user provides,
else *other* a stream terminator which is any non-vector *atom*.

The following serve to define the basic primitives for streams:

(EQ (CDR stream) stream)

This tests if 'stream' is the empty stream.

%L1=(%L1 • %L1) is a representation of an empty stream.

Thus, the empty stream is one which is incapable of emitting anything but the stream itself.

Note: In this section the labels used to convey EQ'ness have scope extending over the entire equation in which they are used.

For example in: g{%L1=(a • b)} = %L1=(c • d) it is meant that %L1 is updated.

Consider the following fast stream which is not empty but is nearly so: %L1=(%L1 • <rfn <string 0 n n> ()>) where a subsequent application of NEXT will produce %L1=(%L1 • %L1). The interpretation of this stream is that it is a unit stream with end-of-line as its last item. The stream itself will serve not only as stream terminator (as *rests*) but also as end-of-line indicator when it appears as *heads*.

CAR, CDR, CONS, RPLACA, RPLACD, etc., can be used as one would expect.

(NEXT stream)

NEXT is a function from streams to streams, which for specific types of streams produces as value stream the argument stream updated. NEXT is most efficient for fast streams. The action of NEXT is defined by the following rules:

next{ %L1=(*heads* • {*other* | %L1}) } → %L1=(%L1 • %L1) .

next{ %L1=(x y • z)} → %L1=(y • z) .

next{%L1=(*heads* • <*rfn* () x ...>)} → *rfn*{%L1}.

next{%L1=(*heads* • %L2=<*rfn* %L3 x...>)}
where %L3=<*string begindex curindex endindex*>

→ *rfn*{%L1} if *curindex* ≥ *endindex*, and *heads* = %L1,

→ %L1=(%L1 . %L2) if *curindex* ≥ *endindex*, and *heads* ≠ %L1,
(This illustrates the production of end-of-line symbols)

otherwise → %L1=(y • %L2=<*rfn* %L3 x...>)
where L3=<*string begindex curindex*+1 *endindex*>
and y = fetchchar{*string* ; *curindex*} .

(WRITE s-exp stream)

write{x;%L1=(y • %L1)} → %L1=(x • NIL) .

write{x;%L1=(y • z)} → %L1=(x y • z)
where z is *other* or a *stream* that is not EQ to %L1.

write{x;%L1=(y • <*rfn* () z...>)} → *rfn*{x;%L1}.

write{x;%L1=(y • %L2=<*rfn* %L3 z...>)}
where %L3=<*string begindex curindex endindex*>

→ %L1=(x • %L2=<*rfn* %L3 z...>)
where %L3=<storechr{ *string*; *curindex*; x} *begindex curindex*+1
endindex>

if *curindex* < *endindex*, and x is a *character*,

otherwise → *rfn*{x;%L1} .

(TEREAD stream)

teread{%L1=(%L1 • y)} → %L1=(%L1 • y) .

For x ≠ %L1 :

teread{%L1=(x • {*other* | %L1})} → %L1=(%L1 • %L1) .

teread{%L1=(x • *stream*)} → %L1=(%L1 • cdr{teread{*stream*}}) .

teread{%L1=(x • %L2=<*rfn* () x...>)} → %L1=(%L1 • %L2) .

teread{%L1=(x • *faststream*)} → %L1=(%L1 • *faststream'*)
where *faststream'* is *faststream* with *curindex* updated to the value of *endindex*.

(TERPRI stream)

terpri{x} = terprix{x;x}.

terprix{x;%L1=(y • {*other* | %L1})} → x .

terprix{x;(y • *stream*)} → terprix{x; *stream*} .

terprix<x; %L1=(y • <*rfn* z...>)} → *rfn*{x;%L1} .

Some Distinguished Streams

LISPIT the console input stream.

LISPIT is a fluid variable with the following initial value:

%L1=(%L1 • < lispitin < nil 0 0 0> asc NIL>)

Where asc=((DEVICE • CONSOLE)(MODE • I)(QUAL • V)).

After the file is activated:

%L1=(item • < lispitin < string beg cur end>asc p-list>)

Where p-list denotes a system dependant I/O control block.

Where lispitin is an input console stream dependant function which is capable of activating the file when the p-list field contains NIL. The function lispitin achieves system independancy by special calls to

system dependant portals for all system dependant computation. Activating this stream consists of:

1. Building an input console p-list in a system dependant manner.

2. Determining the console linelength (also system dependant) and allocating string. Where string is a lisp character vector used to provide an input area for the terminal line. The capacity of which is sufficient to hold the determined maximun input linelength, and the contents-length of which reviels how many it actually holds.

3. Initiallizing beg to 0, cur and end to linelength.

4. Applying lispitin to the now active stream.

When lispitin is applied to an active stream it causes a system dependant console input operation to refill string, reseting string-length to the actual number of characters read, setting end to that number also, and setting beg to zero and cur to one. If the number of characters read was zero the stream becomes:

%L1=(%L1 • <lispitin <" 0 0 0> asc p-list>)
otherwise:
%L1=(c_0 • <lispitin <'$c_0 \ldots c_{end-1}$' 0 1 end> asc p-list>).

LISPOT the console output stream.

LISPOT is a fluid variable with the following intial value:
%L1=(%L1 • <lispotout <NIL 0 0 0> asc p-list>)
where asc=((DEVICE • CONSOLE)(MODE • O)), and p-list=NIL.

lispotout is similar to lispitin except it needs less information to build the p-list. After %L1 is activated by lispotout by write{c;%L1} it becomes:
%L1=(c • <lispotout < string 0 1 end> asc p-list>)
where end is the system dependant preferred console output linelength and string is 'c' . The capacity of string is end characters.

lispotout works in much the same manner as lispitin. One peculiarity of lispotout (and hopefully any output stream which is inactive) occurs when the initial write is in effect a TERPRI.

write{%L1;%L1=(%L1 • <lispotout <NIL 0 0 0> asc NIL>)}
→ %L1=(%L1 • <lispotout <string 0 0 end> asc NIL>)
where string= " but has capacity for 'end' characters.

User Stream Definition Facilities

(DEFIOSTREAM asc linelen position)

DEFIOSTREAM produces as value a fast-stream which interfaces with the real input/output devices.

The actual stream produced is system dependant but the operation of saving a lisp system and bringing it up on another operating system entails the reactivation of all such streams; in which case they may become defined for the new system. The user would have to contrive to have the actual files moved and converted if that were necessary.

The parameters of DEFIOSTREAM are as follows:

'asc' is an a-list, *i.e.* (property...)
where property is:

{(FILE • {('fname' ['ftype' ['fmode']]) | 'dsname'}) |
(DEVICE • CONSOLE) } or,

(RECFM • {F | V}) or,

(MODE • {I | INPUT | O | OUTPUT}) or,

(QUAL •
if CONSOLE input then {S|T|U|V|X}
if CONSOLE output then {LIFO | FIFO | NOEDIT}

The value of the FILE property may be either character string, as indicated, or identifier, in which case the identifier pname is used.

'linelen' is linelength if required, else NIL. For input files, the user supplied linelen is passed to a system dependant portal and the portal gives back a number (possibly the same one) which is used as the actual size of the buffer string which is allocated at activation time. This parameter does not specify a truncation column. For output streams linelen determines both string size and end index.

'position' is a linenumber which defines the starting position if required else NIL.

What follows are some examples of operating system interface streams, their definition and use.

defiostream{asc;72;1}
where asc = ((FILE XXX LISP)(RECFM • V)(MODE • I)).
→ %L1=(%L1 • <filein <NIL 0 72 72> asc NIL 1>)

Comment: Defines an input stream from the file system. The number 72 is the users idea of the length of the longest record. For most operating systems the actual file charactistics will take precedence.

next{%L1=(%L1 • <filein <NIL 0 72 72> asc NIL 1>)}
→ %L1=(c_0 • <filein <%120'c_0...c_{99}' 0 1 100> asc p-list 2>)

where the string '$c_0 \ldots c_{99}$' in this instance has 100 characters but has a capacity for 120 characters because 120 was determined to be the actual longest record of the file.

where p-list is a system dependant I/O control block designation and will not be explained.

This illustrates normal behavior of next when *curindex*≥*endindex* and *heads* is the stream itself, and the line read in is not empty.

Were the first line empty:
next{%L1}→%L1=(%L1 • <filein <" 0 0 0> asc p-list 2>)
and similarly for subsequent empty lines.

On end of file: %L1=(%L1 • %L1) .

defiostream{asc;72;1}
where asc=((FILE YYY LIST)(RECFM • V)(MODE • O))

Comment: Defines an file system output stream. In the case that an old output file exists, its existence is to be ignored as much as possible. The longest record that we wish to write is 72 characters.

Initially the above definition gives rise to:

%L1=(%L1 • <fileout <" 0 0 72> asc NIL 1>) where " has capacity for 72 characters.

write{c_0;%L1=(%L1 • <fileout<" 0 0 72>asc NIL 1>)}
→ %L1=(c_0 • <fileout <'c_0' 0 1 72> asc p-list 1>)

However,
write{%L1;%L1=(%L1 • <fileout<" 0 0 72>asc NIL 1>)}
→ %L1=(%L1 • <fileout <" 0 0 72> asc p-list 2>)

References

1. W. H. Burge. "Stream Processing Functions" *IBM J. Res. Develop.* **19**, 12 (1975).
2. J. E. Stoy and C. Strachey. "OS6—An Experimental Operating System for a Small Computer." *Computer J.* **15**, No. 2. 117 and No. 3. 195 (1972)
3. M. E. Conway. "Design of a Separable Transition-diagram Compiler." *Commun. ACM* **6**, 396 (1963).
4. A. Evans. "PAL—A Language Designed of Teaching Programming Linguistics." *Proc. 23rd ACM Conf.*, 395 (1968).