Treating Data as Source

Jenny Brien

Abstract

Forth has a number of useful facilities for manipulating the interpretation of source text. Unfortunately, they are not always so accessible or useful in the 'interpretation' of other text data. This paper proposes a variation of the Standard that addresses the problem, and also shows a work-around that is illustrated with a simple and extensible XML parser.

What the Standard Forth Interpreter provides

All Standard parsing words use the current input stream. That portion of the input stream currently in memory (the input buffer) is returned by SOURCE, and >IN provides an index to the current position in this buffer, which is updated by the parsing words. In theory, the system can be ignorant as to how the input buffer was filled: REFILL will fill it from the current input stream, if there is one. Input can be re-read by saving and restoring >IN , or by using SAVE-INPUT … RESTORE-INPUT if more than the contents of one input buffer is involved.

When analysing a data stream all this abstraction is not available and we have to write its equivalent from scratch. The input buffer (depending on its type) is `EVALUATE`d, `LOAD`ed or `INCLUDE`d, and that’s all you can do with it. It has to contain valid Forth source. What we need is a way to use an input buffer with any function, not just the Standard text interpreter.

Treating any string like the Input Buffer

Many Forths already supply SOURCE! to set the position and length of the input buffer. It’s simplest and most versatile use is to 'unparse' any parsing word:

```
: EXECUTE-WITH  \ ca u xt -- ; execute parsing word with ca u as its input
     >IN @ >R  SOURCE 2>R >R   SOURCE!
R> EXECUTE  2R> SOURCE!  R> >IN ! ;
```

Example:

```
: $CREATE  \ ca u  -- ; create a header using ca u as the name
  ['] CREATE execute-with ;
```

It's a matter of choice whether you regard this as passing a string to a function or a function to a string!

This will work so long as we are sure the original input buffer has not been over-written. As Michael Gassenenko has pointed out[1] the problem is not with implementing the word itself, but with what happens if SAVE-INPUT or REFILL is attempted before the original input buffer is restored. Gassenenko suggests that implementors should see the input buffer and the refill buffer as separate entities. All input is taken from the input buffer returned by SOURCE. REFILL should not depend on the current value of SOURCE, but only on SOURCE-ID. It should refill a system-supplied refill buffer, and set that to be the SOURCE. Changing the input buffer with SOURCE! would cause interpretation to return to the next line of the original stream when REFILL is called. It follows, then, that changing SOURCE-ID without changing SOURCE would cause interpretation to pass to the next line of the new stream when REFILL is called.

A Wordset for manipulating the input stream

Given assurance of that behaviour, it is possible to specify a complete portable input-manipulating word-set with the addition of two words:

[1] http://forth.sourceforge.net/word/source-store/index.html

```
: SOURCE! \ ca u  -- ;  Set input source. Set >IN to zero
: SOURCE-ID! \ source-id -- ;  Set source id.
```

Nesting and un-nesting sources can then be done in the same style as SAVE-INPUT and RESTORE-INPUT:

```
: SAVE-SOURCE  \  --  xn ..x1 n
  SAVE-INPUT >R SOURCE SOURCE-ID R> 3 + ;

: RESTORE-SOURCE \ xn..x1 n --
  >R IDSOURCE! >R 3- RESTORE-INPUT THROW ;
```

The same definition can get input from any REFILLable source (N>R and NR> move counted sets of parameters to and from an extra stack).

```
: INCLUDE-WITH  \  i*x  source-id xt –- j*x
      save-source N>R
      >R source-id! REFILL THROW
  BEGIN  R@  EXECUTE WHILE REFILL 0= UNTIL THEN
  R> DROP
  NR> restore-source ;
```

By Gassenenko's rule the first REFILL sets SOURCE to the address and length of the input buffer identified by SOURCE-ID. Because INCLUDE-WITH does not open or close files, it will start at wherever the current FILE-POSITION happens to be (i.e. with a newly-opened file it will start with the first line). The function passed to INCLUDE-WITH deals with one line of input per call - though it can deal with more by calling REFILL itself - and exits with a flag to say if it requires any more.

0 SWAP INCLUDE-WITH will provide such a function with input from the terminal input device.

Treating the input buffer like any string

The input buffer is, after all, just another area in memory. Any string-scanning definitions can work with it, so long as we have a pair of words which convert a `>IN` index into a *caddr u* pair and vice versa:

```
\ get the as yet unparsed portion of the input buffer
: PARSE-AREA@     \ -- ca u
  SOURCE >IN @  /STRING ;

\ set the portion of the input buffer still  to be parsed
: PARSE-AREA!     \  ca u -- ; must start within the input buffer!
    DROP SOURCE DROP – 1 CHARS / >IN ! ;
```

This eliminates the awkward difference between parsing source with PARSE and WORD, and parsing strings with other pattern-matching words, when you are left with the *caddr u* of a still unparsed string. Use COMPARE, SEARCH, etc. to write general pattern-matching words with the stack diagram:

```
PatternMatch   \ ca u -- ca1 u1 ca2 u2
```

where ca1 u1 = *string-remaining* and ca2 u2 = *string-matched* and use them in the form:

```
PARSE-AREA@  PatternMatch /dosomething/ PARSE-AREA!
```

Where SOURCE! is available it could of course be used instead of PARSE-AREA! which is in fact just another way of manipulating >IN. It can only modify the size if the parse area by changing its start point, but that is all that is required in this situation (and all that can be achieved within the Standard).

```
: STRING/  \ ca1 u1  u  -- ca2 u2
     SWAP – TUCK – SWAP   ;
```

This is the reverse of /STRING and a useful way to end a PatternMatch word, getting the *string-matched* from the *string-remaining* and the length of the original string.

Data as Source in Standard ANS Forth

The effect of INCLUDE-WITH can be simulated within Standard Forth by simply INCLUDEing a data file. So long as we know what the first word the interpreter encounters will be, we can arrange for it to read and process the rest of the file. A good example of this technique is the method used by Bernd Paysan to add 'active HTML content'[2] -modified here to use PARSE-AREA@ and PARSE-AREA!:

```
:  $> BEGIN parse-area@ S" <$" SEARCH 0= WHILE
          TYPE CR REFILL 0= UNTIL EXIT THEN
          DUP parse-area@ ROT - TYPE
          2 /STRING parse-area! ;

 : <HTML>  $> ;
```

<HTML> causes a HTML file to be interpreted thus: output unchanged anything outside a <$ ..$> bracket, and treat everything inside as Forth source.

JenX - A Simple XML Parser

XML files invariably start with "<?xml" - so that's the word that will do the actual parsing. It reads tags delimited - as in HMTL - by < and > and passes them to a one-shot text interpreter that decides what to do with them, ignoring any that it does not recognise. The definition of JenX itself is therefore quite simple:

```
VALUE DOTAG \  holds the xt of the one-shot interpreter

: JENX  \ xt  ++ ;  parse an XML file using this interpreter
    dotag >R TO dotag INCLUDE >R TO dotag
```

<?XML makes use of two 'stackpads' on which strings as stacked temporarily. One, TAGNAME, is used for the tagnames which are passed to DOTAG, and the other, SCRATCH, holds any text being processed

Stackpad words used

```
SPUSH  \ ca u spad --    ;  push string onto stackpad
SPOP   \ spad -- ca u    ;  pop top string from stackpad
S1     \ spad -- ca u    ;  get top string (no pop)
SDROP  \ spad --         ;  drop top string from stackpad
SNEW   \ spad --         ;  push a zero length string
C+     \ char spad --    ;  append to top string
S+     \  ca u spad --   ;  concat with top string
```

The One-Shot Text Interpreter

A one-shot text interpreter takes a string and performs one action based on the contents of that string, or a common default action if the string is not recognised. It may take the form of a CASE statement, but where the string is a simple word the actions may be defined in a wordlist and a SERVANT may be used

```
: SERVANT  \  wid xt ++ ; defining word for one-shot text interpreters
             CREATE , ,
       DOES>        \ ca u -- ?  ;  do associated action
           >R  2DUP R@ CELL+ @
                 SEARCH-WORDLIST IF
               NIP NIP R> DROP
           EXECUTE ELSE
               R> @ EXECUTE THEN  ;
```

[2] http://www.jwdt.com/~paysan/httpd-en.html

Since the default action (supplied by the xt) still has the string on the stack, it can also be a servant word, and so servants can be stacked in a hierarchy.

```
E.g.     WORDLIST WAITER’S ....
         waiter’s ‘ 2DROP  servant WAITER
         WORDLIST HEADWAITER’S ....
         headwaiter’s ‘ waiter’s servant HEADWAITER


         : CREATION  \ wid -- ; CREATE a word on this wordlist
           GET-CURRENT SWAP SET-CURRENT CREATE SET-CURRENT ;

         : DEF: \ wid -- ; DEFINE a word on this wordlist
           GET-CURRENT SWAP SET-CURRENT : SET-CURRENT ;
```

A Servant Example – dealing with XML entities

```
         WORDLIST CONSTANT ENTITY?

         : CENTITY  \  c ++ ; defining word for single character entities
         ENTITY? CREATION
           C,        \ store the replacement character
         DOES>  \ --   append to scratch stackpad
         C@ scratch c+ ;

         CHAR < CENTITY &LT
         CHAR > CENTITY &GT
         CHAR ' CENTITY &APOS
         CHAR " CENTITY &QUOT
         CHAR & CENTITY &AMP

         : #>C  \ ca u -- c ; convert to ddd or xhhh to char
                \ from Leo Wong
               BASE @ >R
               OVER C@ DUP [CHAR] x = SWAP [CHAR] X = OR IF
               1 /STRING HEX  ELSE DECIMAL THEN  EVALUATE
             R> BASE ! ;

         : UnknownEntity  \ ca u -- ;   try for digits, else append string
               OVER C@  [CHAR] # = IF
               #>C scratch c+ ELSE
               scratch s+ THEN ;

         ENTITY? ' UnknownEntity SERVANT DENT
```

Further defining words can be added later to deal with string substitutions and file inclusions. In this respect, a SERVANT can be seen as an extensible CASE statement.

```
         : DENTS+  \ ca u -- ; append decoded version of string to SCRATCH

               BEGIN [CHAR] & csplit
               scratch s+              \ append text before entity
               DUP WHILE
               [CHAR]  csplit 1 /STRING dent \ append decoded entity
               REPEAT
               2DROP ;
```

A slightly more sophisticated version would use a SERVANT that calls DENT to deal with the Standard Entities, reserving its own wordlist forentities it defines itself by reading the XML file’s DTD.

How <?XML deals with tags

All handling of actual content is done by the xt supplied as a parameter to JenX and stored in DoTag. <?XML just repeatedly parses to the next '<', and places the entire tag on the SCRATCH stackpad. DoTag is passed the address and count of this string, which will be over-written by the next tag.

```
: TILL  \ c -- flag ca u ;   parse string up to char   flag false if char not found
   SOURCE NIP >IN @ - >R PARSE DUP R> = ROT ROT ;

: MACRO  \ Usage: macro <name> <char> <words> <char> (by Wil Baden)
   : char parse  postpone sliteral
     postpone evaluate  postpone
     immediate ;

macro NEXTLINE " WHILE REFILL 0= UNTIL EXIT THEN"
```

When used in conjunction with TILL , NEXTLINE ensures that the intervening code is applied to all input up to, but not including, the delimiting character. If the character is not found before the end of the input stream, then the remainder of the enclosing definition is not executed.

```
: NextTag  \ --  fetch and execute next tag
    Scratch snew
    BEGIN [char] > till
          dents+  \ some tags may contain entities - fetch decoded tag to Scratch
    nextline
    Scratch spop doTag EXECUTE ;

: <?xml  ( -- )
    BEGIN
    BEGIN [char] < till 2DROP nextline
    NextTag
    AGAIN ;
```

<?XML ends when NEXTLINE fails – that is, when input from the file has been exhausted – and returns control to JenX.

Recognising Valid Tag Names

For some simple XML files decoded tag may always be a simple tag name, and the function in DoTag need be nothing more than a SERVANT. Each tag's action is described by a normal Forth word of the same name . This can be the case even for more complex files, if the only tags you want DoTag to act on are simple ones. In all cases, the decoded tag will be overwritten by any word called by DoTag which itself uses the Scratch stackpad, if not by the next execution of NextTag.

There are two other cases which you may need to deal with:

Tags with attribute lists

In this case the tagname is invariably followed by white space. DoTag may call WORDSPLIT to recognise it and pass it on to a SERVANT.

```
: white?  ( c -- ? )  32 > 0= ;

: skip-white  \ ca u -- ca1 u1
     BEGIN DUP WHILE OVER C@ white? WHILE
     1 /STRING REPEAT THEN ;

: scan-white  \ ca u -- ca1 u1
     BEGIN DUP WHILE OVER C@ white? 0= WHILE
     1 /STRING REPEAT THEN ;
```

```
: WORDSPLIT \ ca u -- ca1 u1 ca2 u2 ; remaining-string first-word
     skip-white DUP >R scan-white 2DUP >R string/  ;
```

Processing Instructions and Declarations

These start with "?" and "!" respectively, and depending on the application, may need to be dealt with in a batch or individually. In this case, recognition is based on the characters the string in TAGNAME *starts with* – and can be checked with :

```
MACRO STARTSOF  " >R OVER R> COMPARE TRUE OF "
```

And a CASE statement of the form:

```
\ ca u  from TAGNAME
OVER
  CASE
  S" pattern1" STARTSOF  2DROP action1 ENDOF
(etc)
 \  pass TAGNAME on to WORDSPLIT or a SERVANT
ENDCASE
```

Assume for example that you want to ignore comments:

<!-- does not have to be followed by a space, so defining it as a word won't work.

```
S" !--" startsof doComment endof
```

doComment must ignore everything up to "-->" The comment may span multiple lines, and may enclose tags. If it does not enclose ">" (which is the most likely case) then TAGNAME will already contain the whole comment and we can treat it like any other unknown tag – ignore it. So check for that first.

```
: doComment  \ ca u --
     + 3 CHARS - S" -->" COMPARE IF EXIT THEN
 BEGIN  parse-area@ S" -->" SEARCH 0= WHILE
 2DROP REFILL 0= UNTIL        \ ignore lines until found or eof
     3 /STRING parse-area!    \ parse past -->
```

Matching Tags handle Content

The actual content of XML files is invariably held between matching tag pairs of the form <*name*>… </*name*>. These may be nested inside other tag pairs, so the tagname is saved for matching on the TAGNAME stackpad. TAGNAME will at any point contain, in order, the names of all active tag pairs. That allows it to be used to establish context where tags of the same name may be used by different parents.

I have made the assumption that any content in an inner tag pair without a defined handler should treated as part of the content of the outer pair. That follows naturally from the rule "ignore any unknown tag". The opening tag accumulates content unto the SCRATCH stackpad, processing at will, and executing any tags it meets until the matching closing tag. The space used on SCRATCH is then freed for other tag pairs. The macros TILLMATCH and GETALL encapsulate this behaviour.

```
: GETNAME \ ca u -- ca' u' ; the name of the current tag
      wordsplit 2SWAP 2DROP ;

: MATCHED? \ ca u -- f ; true if current closing tag
  Getname OVER C@ [CHAR] / <> IF 2DROP FALSE EXIT THEN
  1 /STRING DROP TagName s1 COMPARE ;

: OPENTAG \ ca u --   common opening tag initialisation – save name
  GetName Tagname spush  Scratch snew ;
```

```
: CLOSETAG \ ca u --   common closing cleanup – return content
  Tagname sdrop  Scratch spop ;

MACRO TillMatch "  opentag BEGIN BEGIN [char] < till"

MACRO GetAll " nextline parse-area@ matched? 0= WHILE
               NextTag REPEAT closetag "

: PRESERVE-SPACE  \ ca u -- ca u ; of content with space preserved
      TillMatch
      dents+               \ copy decoded string to Scratch
  13 scratch c+            \ add Unix cr
  GetAll   ;

: CONTENT   \ ca u -- ca u ; of content formatted in the default manner
      TillMatch
      BEGIN wordsplit dents+     \ copy decoded string word by word
      BL scratch c+
      DUP 0= UNTIL
      2DROP
  GetAll ;
```

CONTENT will be the word most commonly called when an opening tag is recognised. If the tag has an attribute list which affects processing, it must be dealt with before OPENTAG is called, or else temporarily saved elsewhere.

A Very, Very Simple JenX Application - Output Text of a HTML file

```
Wordlist HTM

HTM DEF:   BODY  TillMatch dents+  Scratch spop TYPE CR
           Getall  2DROP ;

HTM ‘ 2DROP SERVANT HTMTYPE

: SIMPLY  getname htmtype ; \ don’t bother about attributes

: <HTM> <?XML

‘ simply JenX filename
```

And that’s all! The application can be refined later by adding more HTM DEF:s and known entities.