Standardize Strings Now!

M. Anton Ertl[*]
TU Wien

Abstract

This paper looks at the issues in string words: what operations may be required, various design options, and why this has lead to the current state of standardization of string operations that is insufficient in the eyes of many.

1 Introduction

Despite the presence of a string wordset in Forth-94, there are frequent complaints about lack of string support in Forth, and many Forth programmers design their own string library to counter this lack.

2 String operations

This section looks at the string operations present in the language AWK, which is designed for string handling, which gives us an idea of what things string words should be capable of.

AWK is a language that is designed for processing text files, extracting data from them, and outputting the data in some different format. Below we describe GNU AWK (gawk), which offers some features that other AWK variants do not have.

AWK has some language-level capabilities: It splits a file into lines/records (based on a record separator regexp), splitting a line/record into fields (based on a field separator regexp, or a field regexp); it matches lines/records with regexps and uses that to select an action to perform; the action can access the fields through the $\$n$ syntax. AWK also allows easy string concatenation by juxtaposing the two strings, and it supports strings as array indexes.

AWK also provides a number of string functions, which can be divided into several categories:

sorting `asort`, `asorti`

substitution within strings `gensub`, `gsub`, `sub` replace patterns in arbitrary strings, `sprintf` constructs a string from a template.

conversion `strtonum`, `sprintf`

searching `index`, `match`

information `length`

splitting `patsplit`, `split`

substrings `substr`

case conversion `tolower`, `toupper`

3 Design issues

This section discusses the design issues of string words in Forth.

3.1 Desirable Properties

Ease of use One property we would like strings to have is that programming with them is as easy as programming with single or double numbers, without such encumbrances as explicitly managing buffers (including avoiding buffer overflows).

Integration Another nice property is that existing words are useful for dealing with strings. E.g., we can use `2dup 2swap 2over` to handle `c-addr u` type string descriptors on the stack, `2@ 2!` for storing them, and arithmetic words for computing substrings.

As we will see, these two properties are somewhat at odds with each other.

3.2 Allocation

Manual buffer management

Who allocates string buffers, and who frees them?

This issue comes up when generating new strings, such as string concatenation, and is probably the primary issue why we have not found a consensus on a string wordset that includes words for generating new strings (not even concatenation).

One approach is that the word that produces the new string allocates it, e.g.

```
\ s+ ( c-a1 u1 c-a2 u2 -- c-a3 u3 )
dir s" /" file s+ s+ r/o open-file throw
```

[*]Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

The usage looks cute, but it does not free the strings, and therefore is a memory leak. With proper freeing it is no longer so cute:

```
dir s" /" file s+ over >r s+ r> free throw
over >r r/o open-file throw r> free throw
```

This is one reason for disliking this approach, but a stronger one for a significant subset of the Forth community is the use of `allocate`-style allocation itself.

Embedded systems Forths do not necessarily support `allocate`, and even if they have it, one may not want to use it, because of fragmentation or performance concerns. On the other hand, just like embedded users can avoid `allocate` even though it is standardized, they can just as well avoid string creation words that `allocate`, and create strings in the way they do now. One probably won't use Forth as a scripting language on these embedded systems anyway.

Instead of allocating the string buffer in the creating word, one can pass a buffer to the word. This approach is used in `read-line` and `substitute`, and a variant of `s+` with this kind of interface looks as follows:

```
\ s+ ( c-a1 u1 c-a2 u2 c-a3 u3 -- c-a3 u4 n)
create buf1 200 chars allot
create buf2 200 chars allot
dir s" /" buf1 200 s+ 0< abort" buf short"
file buf2 200 s+ 0< abort" buf short"
r/o open-file throw
```

This does not appear attractive, either. A major problem with this approach is that it is possible to provide a too-small buffer, and in general (not for `s+`, but, e.g., for `substitute`), it is hard to know in advance how large the target buffer should be.

Automatic reclamation

So we want to avoid the problems of passing a preallocated buffer as well as the problems of having to free the buffers. Many other languages do this by using garbage collection. We can do that, too, and there is a garbage collector for Forth (written in standard Forth). With garbage collection, we can use the original `s+` usage example.

Requiring garbage collection as part of a string wordset is probably not going to find consensus, however. Garbage collection has a number of disadvantages: It is more complex to implement than explicit deallocation; it is most easily implemented in a stop-the-world fashion, and that does not combine well with real-time systems or multi-threading.

There has been a lot of work on making garbage collection compatible with real-time requirements and multi-threading, but the implementation cost is significant. Also, most (all?) of this work assumes that the compiler and run-time system knows what is an address and what is not; this is generally not possible in Forth.

A practical problem with garbage collection is that, in general, garbage collection has to scan all the data memory, the stacks, and the locals to see which strings are still referenced.

This need can be reduced by always using special words to deal with strings, to keep track of string references. E.g., one might declare all memory storage for string descriptors explicitly, thus avoiding the need to scan all data memory (for dynamically allocated memory, one needs to untrack the memory in some way).

Furthermore, we could have a separate string stack with separate string stack operations, and str@ and str! instructions for accessing string descriptors in memory. This approach has a low integration, though.

If the Forth system knows all the string descriptors, there are additional ways for automatic reclamation: In particular, we can use reference counting (since strings don't contain pointers, the cycle problem of general reference counting cannot occur).

Or, as a variant of that, we can use the following simple string buffer management strategy that ensures that every string only has one reference: copy the string when we copy the descriptor and free the string when we `drop` or overwrite the descriptor (this is inspired by Henry Baker's article on linear logic [Bak94]).

Region-based memory management

A manual reclamation method that is more convenient than `allocate/free` is region-based memory management. The program can create several regions, allocate memory in these regions, and finally free all the memory allocated in a region at once.

You typically collect data into a region if it all becomes garbage and should be freed at (mostly) the same time. E.g., in a compiler you might have a region for stuff that is relevant for a basic block and can be freed after you are done with the block, a region for stuff that is relevant for a colon definition, etc.

One nice feature of regions is that it allows the programmer to decide whether he wants to live with more not-yet-freed garbage or whether he wants to invest more programming effort and have finer-grained regions for less not-yet-freed garbage (up to having the same programming effort as `allocate/free`).

The following example shows a fine-grained use (each of the two memory allocations has a separate region), with the region passed explicitly as a parameter on the stack:

```
\ s+ ( c-a1 u1 c-a2 u2 region-id -- c-a3 u3)
: make-path
  {: dir-a dir-u file-a file-u outer --
     path-a path-u :}
  new-region {: tmp :}
  dir-a dir-u s" /" tmp s+
  file-a file-u outer s+
  tmp free-region ;
```

Here we have two regions: `outer`, and `tmp`. We pass the id of the target region to `s+`, and once we are done with the strings in `tmp`, we free the region.

One problem with this approach is that we have to pass a region-id to any word that returns allocated memory, which causes stack juggling (avoided above by the use of locals); and that additional parameter is needed for every word that generates a string. Instead of passing the region-id explicitly, it can be passed through an implicit *current region* through a context wrapper [Ert11].

Another problem with the example above is that it is not any simpler than explicit deallocation. That's because it does exactly the same thing, and deallocates the intermediate result as soon as possible.

Here is an example where the programmer chooses to let the intermediate result hang around longer, in exchange for easier programming. E.g., if we let the the intermediate result live as long as the final result, and pass the current region implicitly, we can program `make-path` in the ease-of-use way:

```
\ s+ ( c-a1 u1 c-a2 u2 -- c-a3 u3 )
: make-path
  {: dir-a dir-u file-a file-u --
     path-a path-u :}
  dir-a dir-u s" /" file-a file-u s+ s+ ;
: open-path ( dir-a dir-u file-a file-u --)
  new-region dup >r
  ['] make-path with-region
  r/o open-file throw
  r> free-region ;
```

The region management happen at an outer level.

Regions are an interesting idea, but have not made a big impact outside Forth; I guess most go for garbage collection if they want anything more automatic than explicit deallocation. However, given all the problems of general garbage collection, regions may be the way to go for Forth.

One widely available implementation of regions are glibc's obstacks (which offer the additional convenience that every region can be treated as a stack).

3.3 String representation

The favoured string representation in standard Forth is `c-addr u`. It allows representing strings of any length with any content, and you can produce arbitrary substrings without needing to copy the string to a new buffer. The disadvantage of this representation is that it takes two cells on the stack, and dealing with several strings at once can therefore be cumbersome.

The other common string representation in standard Forth is the counted string: The on-stack representation is the address of the count byte; the count byte is followed by the characters of the string. The advantage of this representation is that it needs only one cell on the stack. But it can only represent strings with up to 255 chars, and any substring operation needs to create a new string buffer. Converting from counted to `c-addr u` is easy (`count`), but the other direction is cumbersome. Some people have suggested using cell counts instead of byte counts to get rid of the length limitation.

Some people have proposed using zero-terminated strings (as in C). The on-stack representation is the address of the first character. It can represent strings of arbitrary length that don't contain a NUL char. Substring operations usually need to create a new string buffer (unless the substring is just the tail of the input string). The main advantage is that this string representation makes interfacing to some C functions easier; note that C offers `c-addr u`-compatible versions of many functions in order to be able to deal with arbitrary text; e.g., there is `fputs()` for zero-terminated strings and `fwrite()` for `c-addr u` strings.

If we go with a separate string stack and an in-memory string representation that is only accessed through string words, strings become an abstract data type, and the implementer has a choice of internal string representations. Such a representation may include such things as a reference count.

3.4 Regular expressions

Many scripting languages support searching within strings for a pattern; this is used for selecting among strings, for splitting strings into parts (with the pattern used either as separator or to specify the parts), or for replacing the patterns with replacement strings. The common practice for specifying patterns is regular expressions (regexps); there are some variations of regular expressions, and the Perl 5 variant is probably the most popular one.

All of the uses mentioned above can be implemented with the following regular expression primitive:

```
search-regexp ( c-a1 u1 c-a2 u2 --
   c-a1 u3 c-a4 u4 c-a5 u5 true | false )
```

Search for regexp `c-a2 u2` in string `c-a1 u1`; if the regexp is found, `c-a1 u3` is the substring before the first match, `c-a4 u4` is the first match, and `c-a5 u5` is the rest of the string, and the TOS is true; otherwise return false.

If you use the same regexp several times, it can be more efficient to compile the regular expression into a more readily executed form once, and then use that form repeatedly. An interface for that would be:

```
:regexp  ( c-a2 u2 "name" -- )
```

Compile regular expression `c-a2 u2`, define *name* to perform the action below:

```
name execution: ( c-a1 u1 --
   c-a1 u3 c-a4 u4 c-a5 u5 true | false )
```

Search for regexp `c-a2 u2` in string `c-a1 u1`; if the regexp is found, `c-a1 u3` is the substring before the first match, `c-a4 u4` is the first match, and `c-a5 u5` is the rest of the string, and the TOS is true; otherwise return false.

3.5 Implicit parameters

The `c-addr u` representation leads to words with a lot of stack parameters, e.g., `compare`, `search` and `search-regexp`. This is often cumbersome to work with, and one may want to use some of the techniques for reducing stack depth [Ert11]. In particular, we can use implicit parameters and context-wrappers to get rid of one input and/or output string.

The obvious implicit input parameter is the parse area (`source`), and we can use the context-wrapper `execute-parsing ( addr u xt -- )` to put an input string in the parse area; then we need parsing variants of the words that have too many input strings. E.g., we could have a parsing variant of `search-regexp`:

```
parse-regexp ( c-a2 u2 --
            c-a1 u3 c-a4 u4 true | false )
```

Search for the regexp `c-a2 u2` in the parse area. If a match is found, `c-a4 u4` is the address of the match, and `c-a1 u3` is the string that was skipped before the match was found. The next parse starts right behind the matching string.

For string results, the implicit output parameter is the user output device; i.e., `type` is the implicit-output variant of `move`. The context-wrapper is `>string-execute ( xt -- c-a u )`.

As an example, here we have a program that replaces all the occurences of natural numbers with `<num>`, passing both input and output parameters through a context wrapper.

```
: repl-num1 ( -- )
  begin
    s" [0-9]+" parse-regexp while
      2swap type 2drop ." <num>"
  repeat
  0 parse type ;

: repl-num2 ( c-a u -- )
  ['] repl-num1 execute-parsing ;

: repl-num ( c-a1 u1 -- c-a2 u2 )
  ['] repl-num2 >string-execute ;
```

This code would be a bit tighter with quotations.

4 Conclusion

There are a number of partly conflicting requirements for string packages, in particular

- Ease of use
- Integration with the rest of Forth
- No garbage collection

The various approaches to these problems have led to a large variety of string packages, that cannot be reconciled. Yet, extending the string capabilities of Forth is a much-requested (and, in my case, often-used) feature, so we should standardize additional string capabilities at some point, although the new words will be in parallel to what various string packages offer and ideally make them redundant.

When I started working on this paper, it was unclear to me what the right approach is. Now, it seems to me that the solution is to continue in the direction that the standard string wordset has gone, and add to that:

- Use `c-addr u` as on-stack string representation.
- Add words that create new strings by allocating space for them (e.g., `>string-execute`).
- To make memory reclamation easier, add a region-based memory allocation mechanism (useful not just for strings).

- To reduce the stack depth, use implicit parameters with context-wrappers such as `execute-parsing` and `>string-execute`.
- Add a word or several for matching regular expressions.

References

[Bak94] Henry Baker. Linear logic and permutation stacks — the Forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.

[Ert11] M. Anton Ertl. Ways to reduce the stack depth. In *27th EuroForth Conference*, pages 36–41, 2011.