Recognizer – Dynamically Extend The Forth Interpreter

Matthias Trute

June 24, 2011

Abstract

This article describes a concept for extending the forth text interpreter dynamically to use application specific input data formats at the forth command interpreter. To achieve this, the specific parsing words from the application are used via a standard API[1].

Motivation and Idea

Amforth[2] is a forth system for (very) small systems. Nevertheless it has floating point library as a loadable source module. It would be very handy to enter floating point numbers directly at the command prompt. But: There are huge differences between the "'1000"' and "'1E3"' strings. Not only that the parser needs to deal with it, but the data left of the stack is different: 1Cell for 1000 and 2 cells for the float.

The first idea to extend the interpreter is to straight include the parsing word into the interpreter code and generate a new system that can be flashed to the controller.

This requires source code changes at a very deep level and that could be difficult.

A better way would be a dynamic one. Ideally executing a few simple statements and the forth interpreter magically knows how to deal with the new data formats.

Going further in that direction one may come to the conclusion that the unknown formats already trigger a runtime action: the not-found exception, that can be catched.

But: it is not really simple to catch the exception, deal with it and let the interpreter continue its work just as nothing has happened. Moreover: using an error reporting system for standard operation has many drawbacks.

[1] This article is first published in the Vierte Dimension 2001-02, issued by Forth e.V., Germany. This paper is a modified and translated version of it.

[2] Amforth is a 16bit ITC forth implementation for the AVR Atemga microcontroller family. It is published under GNU GPL at `http://amforth.sf.net/`

In a presentation at the Euroforth 2008 Ulli Hoffmann introduced a new interpreter design with many redefineable hooks. At the first glance a smart idea. If you go into the details it turns out, simple is something different.

Moreover: micro controllers are short on every resources. Any solution has to be small in size and fast. If a user does not need the flexibility, he should not be punished by it. Using conditional source code compilation is not a solution either: more code paths are to be checked.

The concept that is introduced with amforth release 4.3 is called recognizer. The basic idea is taken from a usenet posting by Anton Ertl `https://groups.google.com/group/comp.lang.forth/msg/f70a9ea205b5b75a` during the discussion of number prefixes in 2007:

> Essentially the program has to provide a word (let's call it a recognizer) that takes a string (for the "'word"' that was not found), and returns a flag that indicates whether the string was recognized or not. In addition, the word may do things to the stack below the string/flag (e.g., push literal numbers) or compile things (e.g., literal numbers). So, such a recognizer would have the stack effect:
>
> (i*x c-addr u – j*x f)
>
> [Yes, let's not design counted strings into this interface]
>
> In addition, the interface should support stacking of recognizers, so that one library can provide support for recognizing time syntax, while another library provides support for complex numbers. The program should be able to specify in which order the recognizers should be processed (for cases where the recognized syntax overlaps).

I generalized his ideas to the whole interpreter turning it into an generic tool. Vital parts of the job are handed over to the recognizers.

A recognizer checks one word and tries to understand it. If that is successful, the recognizer will fully work on that word and signals the text interpreter the signal "'I did it"'. Otherwise the recognizer sends the flag "'Sorry, not me."'. In that case the text interpreter will continue with the next recognizer from the list.

Every forth interpreter has 3 default recognizers: `rec-find`, `rec-intnum` and `rec-notfound`. It should be obvious what the first 2 do. The last one does not really understand the word but generates the not-found exception that exits the text interpreter loop and returns to the command prompt.

Implementation

The implementation is straight forward. The existing interpreter is factored into its elements that will be organized in a sorted list.

A conventional interpreter looks like the following code [3]:

[3] see: Hoffmann, Euroforth2008

```
: interpret ( -- )
  BEGIN
    BL WORD DUP COUNT DUP C@
  WHILE ( c-addr )
    FIND ?DUP
    IF OVER STATE @ 0<> =
      IF COMPILE, ELSE
      EXECUTE THEN
    ELSE
      COUNT NUMBER? ?DUP IF 0< IF ?literal
      ELSE
        notfound
      THEN
    THEN
  REPEAT
  DROP ;
```

The sequence of FIND and NUMBER is fixed and cannot be changed. The new interpreter looks like

```
: interpret
  BEGIN
    \ only words with at least 1 character
    BL WORD DUP C@ 0> IF
      \ EE_RECOGNIZER points to an array
      \ in the EEPROM: First field is the number of elements
      \ followed by the elements: the execution tokens
      EE_RECOGNIZERS DUP @E 0 ?DO
        \ The datastack must be very clean: nothing but
        \ the word.
        OVER >R CELL+ DUP >R
        \ read and execute the xecution token of recognizers
        @E EXECUTE
        \ check return flag: either continue
        \ with the next word in the input
        IF R> R> DROP DROP LEAVE THEN
        \ ... or with the next recognizer in the list
        R> R> SWAP
      LOOP
      \ housekeeping
      ?STACK
    REPEAT
    DROP ;
```

This interpreter parses the input to get a single word from it[4]. From the EEPROM a list of execution tokens is read which are called with the word to work on it. Upon exit the flag is consumed to decide whether to proceed with the next word or with the next recognizer.

[4]For pragmatic reason the code is based upon counted strings. The current implementation uses already addr/len pairs to address the word.

That means for words from the dictionary (FIND) that the execution token has to be found in the word lists and depending on the usual criteria either compiled or executed. Numbers are left on the stack or compiled as well.

```
: rec-find
    \ lookup in all active word lists
    FIND
    ?DUP IF
      \ found! now process the XT
      \ immediate?
      0> IF EXECUTE
      ELSE
        STATE @
        IF COMMA, ELSE EXECUTE THEN
      THEN
      \ set the flag for the interpreter
      \ I did it
      -1
    ELSE
      \ clean up and signal
      \ Sorry, not my thing
      DROP 0
    THEN ;
```

The not-found recognizer is simpler. Since it throws an exception, that is not caught by the interpreter it does not need to provide a return flag. Formally it should return an "I did it" flag to the interpreter. But in that case the interpreter would continue with the next word in the input stream not knowing that a more fundamental error has occurred.

```
: rec-notfound
    COUNT TYPE
    -13 THROW
;
```

In the terminal all this looks like (¿ is the system prompt)

```
> ver cr 1 2 + . cr boo
amforth 4.4 ATmega16
3
boo ?? -13 23
>
```

The output `?? -13 23` is generated by the exception catcher in `quit`, that prints the question marks, the exception number an the current `>IN`.

The floating point library mentioned above has a word `>float`, that tries to convert a string to a float number and gives the success/failure flag as well. That makes the float recognizer `rec-float` simple

```
> 123e4 fs.
123e4 ?? -13 6
> : rec-float count >float
 ok if state @ if fliteral then -1 else 0
 ok then ;
 ok
> ' rec-float place-rec
ok
> 123e4 fs.
1.2299999E6 ok
>
```

Management

How to deal with the recognizers? Since the idea is not limited to small systems, the implementation as an EEPROM array could be troublesome on other systems.

I've defined two basic words for the maintenance: `set-recognizer` and `get-recognizer`. They look like and work like `set-order` and `get-order` from the word list wordset.

`Get-recognizer` leaves like `get-order` a list including the number of items on the datastack. This list is saved back and activated with `set-recognizer`.

Some other words may be useful as well. One is the already used `place-rec`. It places a new recognizer into the current list right before the last one. That keeps the standard list intact and makes sure that the not-found recognizer is left as the last one in the list.

```
: place-rec ( xt -- )
    get-recognizer
    \ move away all but the last one
    1-  n>r
    \ place the new recognizer
    swap
    \ and get back all others
    nr>
    \ adjust the number of items
    2 +
    set-recognizer
;
```

Another topic is the `rec-notfound` recognizer itself. It may be included as the final default action into the interpreter code. The current solution allows to drop it and create an at least ambiguous situation.

What else

Beside the floating point library, other uses are possible. At the first hand, they probably will make source code less portable.

First consider name spaces

```
wordlist constant foo
... define word(s) in foo
wordlist constant bar
... define word(s) in bar
```

A recognizer specialized to work with the schema `<wordlist name>::<word name>` could dynamically check the right wordlist for the given wordname.

- `foo::word` lookup vocabulary `foo` after `word` and process it.
- `bar::word` lookup vocabulary `bar` after `word` and process it.

Another scenario is the compilation of large numeric tables. By simply switching the FIND and NUMBER recognizers the almost useless but time consuming dictionary lookups are eliminated.

A third use case are the memory access words currently being defined. Their names are generated formally but the lists is rarely used in full.

Finally

Since version 4.3 amforth implements the recognizers. There are no known problems with it. Work is currently under way to implement locals using recognizers.