Recognizer – Dynamically Extend The Forth Interpreter, Version 2

Matthias Trute

May 30, 2014

Abstract

Recognizers are a programming interface that the forth text interpreter uses to deal with data types. They parse the textual input and convert to preprocessed data. Additionaly they provide methods to deal with the data in the context of the interpreter.

The 2nd generation of recognizers is a redesign based on real code experience. At least two different forth implementations use it: gforth and amforth.

Design Goals

Recognizers are a part of the forth command interpreter. They deal with the textual input and offer methods to deal with it in a generic way.

The primary motivation was to add floating point support to a system that has no support for them without recompiling the system.

Recognizers are proven to work on both big systems (64bit PC) as well as on small system (8 bit microcontrollers).

New Design

The first design [1] did all steps to process the data in one word: parsing the text input and processing it according to STATE. This design turned out to be rather inflexible. The new design uses distingushed words for each of the steps involved including the interpreter actions.

An command interpreter with recognizers is a generic tool that has no information about data types and how to deal with them. It maintains the STATE and is able to identify whitespace delimited words in the input ("SOURCE"). For each of these words, the interpreter consults a list of recognizers until one signals "Success". If no one succeeds, an error message is triggered.

If a recognizer succeeds, it has processed the input word into a format, suitable for further work. A standard interpreter has two recognizers: One to search for the word in the dictionary that may return an execution token and some flags (e.g. immediate). The other standard recognizer deals with integer numbers (both single and double cell precision). These recognizers are close to what the standard words "FIND" and the number conversion words offer.

The interpreter can do exactly two actions with the data: It can interpret or compile them to the dictionary. Each data type provides information how each of these tasks is to be performed.

Parsing

Parsing is the first step from the textual data to work with them. The input is a single word (usually inside SOURCE). A recognizer takes this word and analyzes it. The result is either the "failed" information or "success" together with the data.

The example demonstrates the dictionary lookup recognizer. It searches the dictionary for the word and returns, if found, the execution token and the immediate flags.

The recognizer words in this document follow a naming schema: the prefix REC- denotes the pasing word, the prefix R: is for the type information. The names are rather unimportant, since only their execution tokens are in use.

```
: rec-word ( addr len -- xt +/-1 r:word | r:fail )
   find-name ( -- xt +/-1 | 0 )
   ?dup if
     r:word
   else
     r:fail
   then
;
```

The `r:word` contains the actions, that can be done with an execution token: in the interpret mode (STATE=0) execute it. In compile mode (STATE=1) check the immediate flag and either execute it or compile it to the dictionary. A third method is used to postpone the data (see below).

The three execution tokens are combinded to one with the command "recognizer:" which consumes them under the specified name. The actual implementation can be as simple as an array of three elements or an object with three methods. The only consumer of it is the interpreter itself. For the user the implementation details are completely opaque.

```
\ helper word.
: immediate? 0> ;

  \ every method below get the same
  \ input: ( XT +/-1 -- ) as created in
  \ rec-word above.

  \ execute action
:noname drop execute ;

  \ compile action
:noname immediate? if
    compile,
  else
```

```
    execute
  then ;

  \ postpone action
:noname immediate? if
    postpone postpone
  then
  compile, ;

  \ now 3 XT on the stack
recognizer: r:word
```

The Interpreter

The text interpreter has two things to do. The first is the classical split of SOURCE into whitespace delimited words. The second is to call the recognizers in their respective order. If they return the predefined information "R:FAIL", the next recognizer is called until the recognizer stack is exhausted.

If no recognizer succeeds, the R:FAIL shows its second face: it is a valid processing information for the interpreter. It contains 3 methods too that perform the same: generating an error (usually a not-found exception or a similiar error message).

This makes the interpreter loop quite simple:

```
: interpret
  begin
    parse-name ?dup
    \ no more words?
  while
    ( addr len -- i*x r:table )
    do-recognizers
    state @ if >comp then \ carnal knowledge
    perform
    \ Housekeeping
    ?stack
  until
  \ parse-name always returns addr/len
  \ len is already consumed
  drop
;
```

The word `do-recognizers` does the recognizer iteration. Is is highly implementation depended. Basically it is an iteration over the recognizer stack until one returns something different than R:FAIL.

r:table Methods

The r:table information provides three methods. The interpreter itself uses two of them. The third method is used to perform the "postpone" action to do a delayed compilation.

Interpret

The interpret method performes the intepratation semantics. For numbers it simply leaves them on the stack. An exection token gets executed regardless of the immediate flag.

Compile

The compile action is called when the interpreter is in compile mode. A number is compiled to the dictionary. An execution token is either executed (if the word is immediate) or compiled.

Postpone

Postpone is a special case. It is the only word outside the interpreter that can deal with the information the recognizer returns. The traditional use case for postpone to do compile both normal and immediate words. A postponing of other (literal) data is not defined yet.

Management

To handle the recognizers, two new words are introduced: get- and set-recognizer. They are similiar to get/set-order for wordlists. They take a number of execution tokens from the parsing words and their order in the stack defines the order in which they are called by the interpreter. The standard order is first the rec-word for dictionary lookup, followed by the number recognizer. There is no special not-found recognizer (as in version 1) since the r:fail has taken over this task.

```
' rec-int
' rec-word
2 set-recognizer
```

To define application specific recognizers, the word recognier: is to be used. It gets three exection tokens and a name to identify it. This word is used inside the parsing word to pass the information back to the interpreter

EVALUATE?

Evaluate and friends honor recognizers exactly the same way, the interpreter does.

Current Practice

Currently, two forth systems implement recognizers in the way presented here: gforth and amforth. The only difference between these two is the get/set-recognizer pair. gforth has an additional parameter to support different recognizer stacks. The interpreter uses the one identified by the command "forth-recognizer". amforth does not support this, until a proper use case is presented.

The recognizers itself and the three methods to deal with the results are the same and operate platform independed.

References

[1] Matthias Trute, *Recognizer – Dynamically Extend The Forth Interpreter*, 2011 http://amforth.sourceforge.net/pr/Recognizer-en.pdf

[2] Bernd Paysan, *Recognizers – Customize the Interpreter*, EuroForth 2012 http://www.complang.tuwien.ac.at/anton/euroforth/ef12/papers/paysan-recognizers.pdf

Listings

What follows is an example. It is tested with both systems. The actual implementation is not really useful, but demonstrates the basics.

```
:noname ( morse-bm -- ) ." interpreting"  bin . decimal ;
:noname ( morse-bm -- ) ." compiling "    bin . decimal ;
:noname ( morse-bm -- ) ." postponing "   bin . decimal ;
recognizer: r:morse

: rec:morse
( addr len -- morse-bitmap r:morse | r:fail )
   2>r 0 2r> \ gets morse-bitmap
   bounds ?do
      2* i c@ dup [char] . =
        if drop
        else
        [char] - = if
          1+
        else
          unloop drop r:fail exit
        then
      then
   loop
   r:morse
;
```

Since the parsing word has no explicit dependencies to SOURCE, it can be debugged without including it into the interpreter:

```
(ATmega328P)> s" ---..." rec:morse . bin . decimal
3507 111000  ok
(ATmega328P)>
```

```
mt@Ayla:~$ ./gforth morse-recognizer.frt
Gforth 0.7.9_20140402, Copyright (C) 1995-2013 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type ‘license’
Type ‘bye’ to exit
: bin 2 base ! ;
:12: warning: redefined bin ok
s" ---..." rec:morse . bin . decimal 139979803275096 111000  ok
```

Both forth systems provide more recognizers. Check their respective sources for them. One important one is the introduction of native strings without the hassle of the s" command.