Forth Recognizer -- Request For Discussion

Author: Matthias Trute
Contact: mtrute@web.de

Version: 2
Date: September, 20 2015
Status: Final

Change history

- 2014-10-03 Version 1 - initial version.
- **2015-05-17 Version 2 - extend rationale.**
    - 2015-05-17 discuss tick (')
    - 2015-06-05 editorial changes.
    - 2015-06-07 2-method API with code examples
    - 2015-06-13 example implementation of ' (tick)
    - 2015-06-22 add tick (') and bracket-tick ['] added to the normative part.
    - 2015-07-11 more about flags and R:FAIL, cross compiler
    - 2015-08-23 not-found recognizer, multi-word recognizer, document structure
    - 2015-09-20 finalized v2

Background

I'm working on a Forth for 8-bit micro-controllers for almost 10 years now (amforth.sf.net). It is not only a useful tool for serious work but a nice playground to experiment with Forth too.

In 2011 my Forth got a floating point library. Since a micro-controller is (was) a resource constrained system it is not an option to include it permanently. It has to be a loadable module. Therefore I needed a way to keep the core system small but at the same time able to fully handle the new numbers. All but one problem were easy to fix. Only adding the number format to the Forth interpreter turned out to be serious one. I searched the net for ways to extend the Forth interpreter. What I found was having many hooks in the interpreter (U. Hoffman, Euroforth 2008) or a conditional re-compile of the sources with an autotool/configure like build system. Nothing really convinced me or my users. While googling I stumbled across the number parsing prefix discussion in c.l.f in 2007. The ideas sketched there looked promising so I stopped searching and started with them to invent my own solution.

I changed the Forth interpreter into a dumb tool, that delegates all data related work to modules, which can be changed at run-time. That made it possible to load the FP library into a running system that afterwards was able to deal with the new numbers like native ones. Surprisingly the new system had no disadvantages in speed or size compared the old one, something I consider very important on a micro-controller.

Shortly thereafter, Bernd Paysan got interested in what I did (we have regular IRC sessions on Forth topics) and started to implement recognizers in gforth. He suggested changes that further simplified my concept and made it more flexible.

By now we reached a point that justifies the public review. There are two very different Forth's available (both GPL'ed) that implement recognizers. A third implementation is in the proposal (public domain).

A recognizer written for one Forth works without modification for the other ones too. The words used to actually implement a recognizer (mostly string processing) need to be available of course. E.g. I wrote a

recognizer for time stamp strings with gforth that converts the hh:mm:ss notation into a double cell number for the seconds since midnight. The code runs on amforth too. Gforth is a 64-bit system on the PC, amforth a 16-bit system on an 8-bit micro-controller (hence the double numbers). With that, something like

```
: test 01:00:01 d. ."  seconds since midnight" ; ok
test 3601 seconds since midnight ok
01:01:00 01:00:01 d+ d. 7261 ok
```

is possible. Similarly strings: everything that starts with a " is a string until the closing " is reached. Further string handling get the addr/len without the enclosing ".

```
: test "A string" type ; ok
test A string ok
" Another string" type ok  Another string
```

Another use case are name-spaces with word lists, without touching ORDER:

```
: test i2c.begin i2c.sendbyte i2c.end ;
```

where begin/sendbyte/end are words from the word-list identified with i2c (a constant with the wid). The recognizer splits the word at the first dot and uses the left sub-word to get the a word-list. In that word-list it searches with the remaining string and handles the result just like an ordinary dictionary search: interpret, compile (or not found).

Implementations for these examples are available in the respective Forth systems.

Problem

The Forth compiler can be extended easily. The Forth interpreter however has a fixed set of capabilities as outlined in section 3.4 of the standard text: Words from the dictionary and some numbers.

It's not easily possible to use the Forth text interpreter in an application or system extension context. The building blocks (FIND, COMPILE,, >NUMBER etc) are available but there is a gap between them and what the Forth interpreter does. Applications need to use either additional system provided and system specific intermediate words (if available) or have to re-invent the wheel to use e.g. numbers with a sign or hex numbers with the $ prefix.

Some Forth interpreters have ways to add new data types. That makes it possible to use a loadable library to implement new data types to be handled like the built-in ones. An example are the floating point numbers. They have their own parsing and data handling words including a stack of their own.

To actually handle data in the Forth context, the processing actions need to be STATE aware. It would be nice if the Forth text interpreter, that maintains STATE, is able to do the data processing without exposing STATE to the data handling methods. For that the different methods need to be registered somehow.

Whenever the Forth text interpreter is mentioned, the standard words EVALUATE (CORE), INCLUDE-FILE (FILE), INCLUDED (FILE), LOAD (BLOCK) and THRU (BLOCK) are expected to act likewise.

Solution

The monolithic design of the Forth interpreter is factored into three major blocks: First the interpreter. It maintains STATE and organisers the work. Second the actual data parsing. It is called from the interpreter and analyses strings (usually sub-strings of SOURCE) if they match the criteria for a certain data type. These parsing words are grouped as a stack to achieve an order of invocation. The result of the parsing words is handed over by the interpreter to data specific handling methods. There are three different methods for each data type depending on STATE and to POSTPONE the data.

The combination of a parsing word and the set of data handling words to deal with the data is called a recognizer. There is no strict 1:1 relation between the parsing words and the data handling sets. A data handling set for e.g. single cell numbers can be used by different parsing words.

Proposal

XY. The optional Recognizer word set

XY.1 Introduction

The algorithm of the Forth text interpreter as described in section 3.4 is modified as follows. All subsections of 3.4 apply unchanged.

a. Skip leading spaces and parse a name. Leave if the parsing area is empty.

b. For each element of the recognizer stack, starting with the top element, call its parsing method with the sub-string "name" from step a).

   Every parsing method returns an information token and the parsed data from the analysed sub-string if successful. Otherwise it returns the system provided failure token `R:FAIL` and no further data.

   Continue with the next element in the recognizer stack until either all are used or the information token returned from the parsing word is not the system provided failure token `R:FAIL`.

c. Use the information token and do one of the following

   1. if interpreting execute the interpret method associated with the information token.
   2. if compiling execute the compile method associated with the information token.

d. Continue with a)

A recognizer consists of two elements: a parsing word (`REC:TABLE`) and one or more information tokens returned by the parsing words that identify the parsed data and provide methods to perform the various semantics of the data (interpret, compile and postpone). A parsing word can return different information tokens. A particular information token can be used by different parsing words.

There is a system provided information token called `R:FAIL`. It is used if no other token is applicable. This failure token is associated with the system error actions if used in step c).

The parsing word of a recognizer has the stack effect

```
REC:TABLE ( addr len -- i*x R:TABLE | R:FAIL )
```

"addr/len" is a sub-string provided by the Forth text interpreter inside `SOURCE`. The parsing word must not change the string content. It can change `>IN` however.

"i*x" is the result of the text parsing of the string found at "addr/len". `R:TABLE` is the information token that the interpreter uses to execute the interpret, compile or postpone actions for the data "i*x".

All three methods are called with the "i*x" data as left by the parsing word.

```
R:TABLE:METHOD ( ... i*x -- j*y )
```

They can have additional stack effects, depending on what `R:TABLE:METHOS` actually does.

The data items "i*x" don't have to be on the data stack, they can be at different places, when applicable. E.g. floating point numbers have a stack of their own. In this case, the data stack contains the `R:TABLE` information only.

The names `R:TABLE`, `REC:TABLE` and `R:TABLE:METHOD` don't have to actually exist, except the `R:FAIL` name.

A Forth system shall provide recognizers for integer numbers (both single and double precision) and the word look-up in the dictionary. They shall be ordered in a way that the word look-up is called first followed by the one(s) for numbers.

There shall be at least 4 recognizer slots available for application use.

XY.2 Additional terms and notations

Information token: A single cell number. It identifies the data type and a method table to perform the data processing of the interpreter. A naming convention suggests that the names start with `R:`.

Recognizer: A combination of a text parsing word that returns information tokens together with parsed data if successful. The text parsing word is assumed to run in cooperation with `SOURCE` and `>IN`. A naming convention suggests that the names start with `REC:`.

XY.3 Additional usage requirements

XY.3.1 Environment Queries

Obsolete.

XY.4 Additional documentation requirements

XY.4.1 System documentation

XY.4.1.1 Implementation-defined options

No additional options.

XY.4.1.2 Ambiguous conditions

- An empty recognizer stack.
- Changing the content of the parsed string during parsing.

XY.4.2 Program documentation

- No additional dependencies.

XY.5 Compliance and labelling

The phrase "Providing the Recognizer word set" shall be appended to the label of any Standard System that provides all of the Recognizer word set.

XY.6 Glossary

XY.6.1 Recognizer words

' ("<spaces>name" -- XT) RECOGNIZER

Change the behaviour of 6.1.0070 ' (CORE) to use the recognizer stack for the dictionary look-ups. An ambiguous condition exists if `name` is not a word with interpretation semantics (e.g. a number).

DO-RECOGNIZER (addr len -- i*x R:TABLE | R:FAIL) RECOGNIZER

Apply the string at "addr/len" to the elements of the recognizer stack. Terminate the iteration if either a recognizer returns a information token that is different from `R:FAIL` or the stack is exhausted. In this case return `R:FAIL`.

"i*x" is the result of the parsing word. It may be on other locations than the data stack. In this case the stack diagram should be read accordingly.

There is an ambiguous condition if the recognizer stack is empty.

GET-RECOGNIZERS (-- rec-n .. rec-1 n) RECOGNIZER

Return the execution tokens `rec-1 .. rec-n` of the parsing words in the recognizer stack. `rec-1` identifies the recognizer that is called first and `rec-n` the word that is called last.

The recognizer stack is unaffected.

MARKER ("<spaces>name" --) RECOGNIZER

Extend `MARKER` to include the current recognizer stack in the state preservation.

R:FAIL (-- R:FAIL) RECOGNIZER

A constant cell sized information token with two uses: first it is used to deliver the information that a specific recognizer could not deal with the string passed to it. Second it is a predefined information token whose elements are used when no recognizer from the recognizer stack could handle the passed string. These methods provide the system error actions.

The actual numeric value is system dependent and has no predictable value.

RECOGNIZER: (XT-INTERPRET XT-COMPILE XT-POSTPONE "<spaces>name" --) RECOGNIZER

Skip leading space delimiters. Parse name delimited by a space. Create a recognizer information token "name" with the three execution tokens. The implementation is system dependent.

The words for XT-INTERPRET, XT-COMPILE and XT-POSTPONE are called with the parsed data that the associated parsing word of the recognizer returned. The information token itself is consumed by the interpreter.

SET-RECOGNIZERS (rec-n .. rec-1 n --) RECOGNIZER

Set the recognizer stack to the recognizers identified by the execution tokens of their parsing words `rec-n .. rec-1`. `rec-1` will be the parsing word of the recognizer that is called first, `rec-n` will be the last one. If n is not a positive number, an ambiguous condition is met. A minimum recognizer stack shall include the words for dealing with the dictionary and integer numbers.

['] ("<spaces>name" --) RECOGNIZER

Modify 6.1.2610 `[']` (bracket-tick, CORE-EXT) to use the recognizer stack for dictionary look-ups. An ambiguous condition exists if name is not found or has no interpretation semantics (e.g. a number)

XY.7 Reference Implementation

The code has as little as possible dependencies (basically only CORE and CORE EXT). The implementations in gforth and amforth differ and use highly system specific strategies. The code has been tested on gforth 0.7.0.

```
\ create a simple 3 element structure
: RECOGNIZER: ( XT-INTERPRET XT-COMPILE XT-POSTPONE "<spaces>name" -- )
   CREATE SWAP ROT , , ,
;

\ system failure recognizer
:NONAME -13 THROW ; DUP DUP RECOGNIZER: R:FAIL

\ helper words to decode the data structure created by
\ RECOGNIZER: The knowledge they represent is used inside
\ POSTPONE and the text interpreter only.
: _R>POST ( R:TABLE -- XT-POSTPONE ) CELL+ CELL+ @ ;
: _R>COMP ( R:TABLE -- XT-COMPILE  )       CELL+ @ ;
: _R>INT  ( R:TABLE -- XT-INTERPRET)             @ ;

\ contains the recognizer stack data
\ first cell is the current depth.
10 CELLS BUFFER: rec-data
0 rec-data ! \ empty stack
```

```
: SET-RECOGNIZERS ( rec-n .. rec-1 n -- )
   DUP rec-data !
   BEGIN
     DUP
   WHILE
     DUP CELLS rec-data +
     ROT SWAP ! 1-
   REPEAT DROP
;

: GET-RECOGNIZERS ( -- rec-n .. rec-1 n )
   rec-data @ rec-data
   BEGIN
     CELL+ OVER
   WHILE
     DUP @ ROT 1- ROT
   REPEAT 2DROP
   rec-data @
;

: DO-RECOGNIZER   ( addr len -- i*x R:TABLE | R:FAIL )
    rec-data @
    BEGIN
      DUP
    WHILE
      DUP CELLS rec-data + @
      2OVER 2>R SWAP 1- >R
      EXECUTE DUP R:FAIL <> IF R> DROP 2R> 2DROP EXIT THEN DROP
      R> 2R> ROT
    REPEAT
    DROP 2DROP R:FAIL
;
```

`POSTPONE` is outside the Forth interpreter:

```
: POSTPONE (  "<spaces>name" -- )
  PARSE-NAME DO-RECOGNIZER
  _R>POST    \ get the XT-POSTPONE from R:TABLE
  EXECUTE
; IMMEDIATE
```

Implementing `'` requires system specific information. The rationale section contains example code. `[']` is straight forward

```
: ['] ' POSTPONE LITERAL ; IMMEDIATE
```

A.XY Informal Appendix

A.XY.1 Forth Text Interpreter

The Forth text interpreter turns into a generic tool that is capable to deal with any data type. It maintains `STATE` and calls the data processing methods according to it.

```
: INTERPRET
  BEGIN
      PARSE-NAME DUP
  WHILE
      DO-RECOGNIZER ( addr len -- i*x R:TABLE | R:FAIL)
      STATE @ IF _R>COMP ELSE _R>INT THEN \ get the right XT from R:*
      EXECUTE \ do the action.
      ?STACK  \ simple housekeeping
  REPEAT 2DROP
;
```

A.XY.2 Example Recognizer

The first example looks up the dictionary for the word and returns the execution token and the header flags if found. The data processing is the usual interpret/compile action. The Compile actions checks for immediacy and act accordingly. A portable postpone action is not possible. Amforth and gforth do it in a system specific way.

```
\ find-word is close to FIND but takes addr/len as input
256 BUFFER: find-word-buf
: place ( c-addr1 u c-addr2 ) 2DUP C! CHAR+ SWAP MOVE ;
: find-word ( addr len -- xt +/-1 | 0 )
    find-word-buf place find-word-buf
    FIND DUP 0= IF NIP THEN ;

: immediate? ( flags -- true|false ) 0> ;
:NONAME ( i*x XT flags -- j*y )  \ INTERPRET
  DROP EXECUTE ;
:NONAME ( XT flags -- )          \ COMPILE
  immediate? IF COMPILE, ELSE EXECUTE THEN ;
:NONAME ( XT flags -- )          \ POSTPONE
  \ system specific implementation required.
RECOGNIZER: R:WORD

: REC:WORD ( addr len -- XT flags R:WORD | R:FAIL )
   find-word ( addr len -- XT flags | 0 )
   ?DUP IF R:WORD ELSE R:FAIL THEN ;

\ prepend the find recognizer to the recognizer stack
GET-RECOGNIZERS ' REC:WORD SWAP 1+ SET-RECOGNIZERS
```

The second example deals with floating point numbers. The interpret action is a do-nothing since there is nothing that has to be done in addition to what the parsing word already did. The compile action takes the floating point number from the FP stack and compiles it to the dictionary. Postponing numbers is not defined, thus the postpone action here is printing the number and throwing an exception.

```
:NONAME ;                    ( -- ) (F: f -- f) \ INTERPRET
:NONAME POSTPONE FLITERAL ; ( -- ) (F: f -- )  \ COMPILE
:NONAME FS. -48 THROW ;      ( -- ) (F: f -- )  \ POSTPONE
RECOGNIZER: R:FLOAT

: REC:FLOAT ( addr len -- (F: -- f ) R:FLOAT | R:FAIL )
  >FLOAT IF R:FLOAT ELSE R:FAIL THEN ;
```

```
\ append the float recognizer to the recognizer stack
' REC:FLOAT GET-RECOGNIZERS 1+ SET-RECOGNIZERS
```

Experience

First ideas to dynamically extend the Forth text interpreter were published in 2005 at comp.lang.forth by Josh Fuller and J Thomas: Additional Recognizers?

A specific solution to deal with number prefixes was roughly sketched by Anton Ertl at comp.lang.forth in 2007 with https://groups.google.com/forum/#!msg/comp.lang.forth/r7Vp3w1xNus/Wre1BaKeCvcJ

There are a number of specific solutions that can at least partly be seen as recognizers in various Forth's:

- prefix-detection in ciforth
- W32Forth uses its "chain" concept to achieve similar effects.
- various commercial Forth's seem to have ways to extent the interpreter.

A first generic recognizer concept was implemented in amforth version 4.3 (May 2011). The design presented in this RFD is implemented with version 5.3 (May 2014). gforth has recognizers since 2012, the ones described here since June 2014.

Existing recognizers cover a wide range of data formats like floating point numbers and strings. Others mimic the back-tick syntax used in many Unix shells to execute OS sub-process. A recognizer is used to implement OO notations.

Most of the small words that constitute a recognizer don't need a name actually since only their execution tokens are used. For the major words a naming convention is suggested: `REC:<name>` for the parsing word of the recognizer "name", and `R:<name>` for the information token word created with `RECOGNIZER:` for the data type "name".

There is no `REC:FAIL` that would be the companion of the system provided `R:FAIL`. It's simply

```
: REC:FAIL ( addr len -- R:FAIL )
  2DROP R:FAIL ;
```

That way, `REC:FAIL` can be seen as the parsing word of the recognizer that is always present as the last one in the recognizer stack and that cannot be deleted.

A Forth system that uses recognizers in the core has words for numbers and dictionary look-ups. These recognizers are useful for other data formats and use cases as well. They shall be named identically as shown in the table:

Name	Stack items	Comment
`R:NUM`	`( -- n R:NUM)`	single cell numbers, based on `>NUMBER`
`R:DNUM`	`( -- d R:DNUM)`	double cell numbers, based on `>NUMBER`
`R:FLOAT`	`( -- f R:FLOAT)`	floating point numbers, based on `>FLOAT`
`R:WORD`	`( -- XT flags R:WORD)`	words from the dictionary, `FIND`
`R:NAME`	`( -- NT R:NAME)`	words from the dictionary, with name tokens NT

The matching parsing words should be available as

Name	Stack effect
`REC:NUM`	`( addr len -- n R:NUM | d R:DNUM | R:FAIL )`
`REC:FLOAT`	`( addr len -- f R:FLOAT | R:FAIL )`

REC:WORD	(addr len -- XT flags R:WORD | R:FAIL)
REC:NAME	(addr len -- NT R:NAME | R:FAIL)

Test cases

The hardest and ultimate test case is to use the interpreter with recognizers enabled. Some parts can be tested separately, however.

```
T{ GET-RECOGNIZERS SET-RECOGNIZERS -> }T
T{ GET-RECOGNIZERS SET-RECOGNIZERS GET-RECOGNIZERS -> GET-RECOGNIZERS }T
T{ s" unknown word" DO-RECOGNIZER  -> R:FAIL }T
```

The system provided recognizers, if available, work as follows:

```
T{ s" 1234"  DO-RECOGNIZER -> 1234  R:NUM   }T
T{ s" 1234." DO-RECOGNIZER -> 1234. R:DNUM  }T
T{ s" 1e3"   DO-RECOGNIZER ->  1e3  R:FLOAT }T
T{ s" DO-RECOGNIZER" DO-RECOGNIZER -> ' DO-RECOGNIZER -1 R:WORD }T
```

Extended Rationale from the discussion of Version 1

There was an almost common agreement that recognizers shall replace the default command interpreter behaviour if provided by the system implementer. Andrew Haley suggests that recognizers should be used as a least resort tool only if the standard text interpreter cannot deal with the input data. That means that the interpreter will always handle the dictionary searches and the number checks itself and only if they fail activates the recognizer stack. This leaves the interpreter untouched but blocks the full flexibility. It would be impossible to change the first two steps: dictionary look-up and the (limited) number recognition. The final wordings may find a solution for that. The majority questions the usefulness of such a 2 class interpreter design however.

Name Tokens

Name Tokens (nt) are part of the Forth 2012 Programming Tools word set.

The words found in the dictionary with FIND return the execution token and the immediate flag. Using the Programming Tools word set, the dictionary look-up can be made based on `TRAVERSE-WORDLIST` with a recognizer called e.g. `REC:NAME ( addr len -- nt R:NAME|R:FAIL)`. The major difference to `FIND` is that all header information is available to handle the token:

```
:NONAME NAME>INTERPRET EXECUTE ; ( nt -- ) \ interpret
:NONAME NAME>COMPILE EXECUTE ;   ( nt -- ) \ compile
:NONAME NAME>COMPILE SWAP POSTPONE LITERAL COMPILE, ; \ postpone
RECOGNIZER: R:NAME
```

The actual `REC:NAME` is slightly more complex and requires usually system knowledge.

```
\ the analogon to search-wordlist
: search-name ( addr len wid -- nt | 0 )  .... ;

\ a single wordlist is checked
: (rec:name)    ( addr len wid -- nt r:name | r:fail )
  search-name
  dup if
```

```
    r:name
  else
    drop r:fail
  then
;

\ checks only the forth standard word-list
: rec:name ( addr len -- nt r:name | r:fail)
  forth-wordlist (rec:name)
;
```

To handle a set of word-lists like the order stack additional steps may be necessary.

Search Order Word Set

A large part of the Search Order word set is close to what recognizers do while dictionary searches. The order stack can be seen as a subset of the recognizer stack.

The words dealing with the order stack (`ALSO`, `PREVIOUS`, `FORTH`, `ONLY` etc) may be extended/changed to handle the recognizer stack too/instead. On the other hand, `ALSO` is essentially `DUP` on a different stack. `ONLY` and `FORTH` set a predefined stack content. With the `GET/SET-RECOGNIZERS` words all changes can be prepared on the data stack with the usual data stack words.

A complete redesign of the Search Order word set affects many programs, worth an own RFD. The common tools to actually implement both recognizer and search order word sets may be useful for themselves.

Completely unrelated is `SET/GET-CURRENT`. Recognizers don't deal with the places, new words are put into. Possible changes here are not considered part of the recognizer word set proposal.

GET/SET-RECOGNIZERS

These commands can create a deep data stack usage. They are modelled after the well established `GET-/SET-ORDER` and `N>R/NR>` word pairs.

An alternative solution are words following `>R` and `R>`. Likewise a `>RECOGNIZER` would put the new item on the top of the recognizer stack. Since this element is processed first in `DO-RECOGNIZER`, this action prepends to the recognizer stack. Having the recognizer loop acting the other way (bottom up) is confusing and therefore not an option too. Furthermore I expect that most changes to the recognizer stack take place at the *end* (bottom) of it appending a new recognizer. There is no commonly agreed way to access a stack at its bottom. Even more difficult is an insert or remove operation of a recognizer in the middle. Again the standard data stack words are the simplest way to do it. Since the recognizer stack is smaller than the data stack and stack changes are expected to happen seldom the proposed solution is considered the simplest solution.

Stack or List

The recognizers use a stack to define their order. Technically an ordered list can do the same. Since Forth is a stack oriented language, a stack is suggested. The same reason applies to the chains, which are essentially a combination of lists and defers.

POSTPONE and '

Adding the POSTPONE method has been seen as overly complex. At least with the current standard text it is necessary however. One reason is that POSTPONE has a lot of special cases which cannot be implemented without system knowledge. The postpone method carries this information for all data types. Recent discussions indicate that this may be solved cleanly in a future version of Forth, until this discussion is finished, a separate postpone action is the only way to implement what recognizers can achieve.

' (tick) and its companion ['] (bracket-tick) is less difficult. It is common practice that the sequence ' foo execute does the same as calling foo directly (in interpret mode). Now consider a special recognizer that searches an otherwise hidden word-list (think of name spaces). Words from it may be interpreted and compiled without problems, but could not be found with '. Therefore it is desirable to use the recognizer stack here too. The difficulty here is to decide whether a recognised item is an executable "tick-able" word. E.g. numbers and compile-only words are not.

Implementation requires system specific knowledge. The following depends on R:WORD to work.

```
: executable? ( R:TABLE -- f )
          _R>INT \ get the interpretation action for the given token
   R:WORD _R>INT \ get the system specific interpret action
   =
;

: ' ( "<spaces>name" -- XT )
  PARSE-NAME DO-RECOGNIZER
  executable? 0= IF
    \ call the system specific error action "invalid tick"
    -13 THROW
  THEN
  DROP \ remove the immediate flag
  \ the XT from the R:WORD result set is left
;
```

2-Method API

Anton Ertl suggested an alternative implementation of recognizers. Basically all text data is converted into a literal at parse time. Later the interpreter decides whether to execute or compile the literal data, depending on STATE. POSTPONE is a combination of storing the literal data together with their compile time action.

```
interpretation: conv final-action
compilation:    conv literal-like postpone final-action
postpone:
      conv literal-like postpone literal-like postpone final-action
```

The conv-action is what is done inside the DO-RECOGNIZER action (REC:* words) and the literal-like and final-action set replaces the proposed 3 method set in R:*. It is not yet clear whether this approach covers the same range of possibilities as the proposed one or may solve the tick-problem mentioned above. Another side effect is that postponing literals like numbers becomes possible without further notice.

For simple use cases (literals) it's possible to automatically convert this approach into the 3-method API (Anton Ertl and Bernd Paysan):

```
: rec-methods {: literal-xt final-xt -- interpret-xt compile-xt postpone-xt :}
  final-xt
```

```
  :noname literal-xt compile, final-xt ]] literal compile, ; [[ dup >r
  :noname literal-xt compile, r> compile, postpone ;
;
```

With that command, the standard number recognizer can be rewritten as

```
\ numbers
:noname ; \ final-action do nothing
' literal \ literal-action
rec-methods recognizer: r:num
```

Anton Ertl writes in comp.lang.forth:

> If you define recognizers through these components, you don't need to specify the three components, in particular not a POSTPONE action; and yet POSTPONEing literals works as does any other POSTPONEing of recognizers. With that, one might leave it up to systems whether they support POSTPONEing recognizers or not.
>
> Disadvantage: Does not combine with doing the dictionary look-up as a recognizer for immediate words:
>
> If you make the immediate word a parse-time action with a noop for literal-like and noop for run-time, it works correctly for interpretation and compilation, but not for POSTPONE. And anything else is even further from the desired behaviour. One could enhance this scheme to support immediate words correctly, but I don't see a clean way to do that.
>
> So there seems to be a choice:
>
> 1. Compose the behaviour of recognizers of these components, but do not treat the dictionary as a recognizer.
> 2. Treat the dictionary as a recognizer, but build recognizers from interpretation, compilation, and postponeing behaviour.

A complete reference implementation does not exist, many aspects were published at comp.lang.forth by Jenny Brien.

Flags, `R:FAIL` or Exceptions

The `R:FAIL` word has two purposes. One is to deliver a boolean information whether a parsing word could deal with a word. The other task is the method table of for the interpreter to actually handle the parsed data, this time by generating a proper error message and to leave the interpreter. While the method table simplifies the interpreter loop, the flag information seems to be odd. On the other hand a comparison of the returned `R:*` token with the constant `R:FAIL` can be easily optimised.

A completely different approach is using exceptions to deliver the flag information from `DO-RECOGNIZER` to its callers. Using them requires the exception word set, which may not be present on all systems. In addition, an exception is a somewhat elaborate error handling tool and usually means than something unexpected has happened. Matching a string to a sequence of patterns means that exceptions are used in a normal flow of compare operations.

That `R:FAIL` is used in two ways is an optimisation. The flag information can be carried with the equation `R:* R:FAIL =` as well.

That there is no final R:FAIL in the recognizer stack is an optimisation too. Earlier versions of the recognizer concept did have such a bottom element. It turned out that it caused a lot of trouble. If it got deleted, the interpreter loop did not recognise this as an error and crashed without further notice. Adding a check for an empty recognizer stack was more code. The second argument against is that adding a recognizer to the recognizer becomes more complex since there is a bottom element, that should be kept, essentially making appending a recognizer always an insert-in-the-middle action.

Multiword Parsing

The RfD suggests that the input stream is split into white-space delimited words as part of the general text interpreter. The parse actions of the recognizers get these words only. A recognizer that deals with "sentences" (multiple words) needs more specifications:

- The input for the text interpreter comes from `SOURCE` and is managed with `>IN`. Systems are free to make a copy of the word before calling the parsing words from the recognizer. A multi-word recognizer nevertheless needs to (read-only) access the `SOURCE` buffer directly.
- It is possible that the closing syntactic element of the multi-word sentence is not within the current `SOURCE` content. A `REFILL` may be necessary to get it. Since that may be troublesome in the long run, the closing element shall be in the same `SOURCE` buffer as the opening one.

There is no dependency on `SOURCE` and `>IN` for the simple, single-word recognizer. To completely eliminate these dependencies, the information from `SOURCE` and `>IN` need to become part of the input and return data of the parsing words which would create a lot of stack traffic with no benefit for the common use case of a single word.

Additional Remarks

This sections contains less related stuff. It is more or less a collection of ideas around recognizers. The topics here are not necessarily discussed openly already, but seem to be worth mentioned.

Stateless interpreter

An implementation of the interpreter without an explicit `STATE`. For legacy applications a `STATE` variable is maintained but not used.

The code depends on `DEFER` and `IS` from CORE EXT. Similar code can be found in gforth and win32forth.

```
\ legacy state support
VARIABLE STATE
: on ( addr -- )  -1 SWAP ! ;
: off ( addr -- )  0 SWAP ! ;

\ the two states of the interpreter
: (interpret-i) _R>INT EXECUTE ;
: (interpret-c) _R>COMP EXECUTE ;
DEFER (interpret) ' (interpret-i) IS (interpret)

\ switch interpreter modes
: ] STATE on ['] (interpret-c) IS (interpret) ;
: [ STATE off ['] (interpret-i) IS (interpret) ; IMMEDIATE

: interpret
    BEGIN
      PARSE-NAME DUP \ get something
    WHILE
      DO-RECOGNIZER  \ analyze it
      (interpret)    \ act on it
      ?stack         \ simple housekeeping
    REPEAT 2DROP
 ;
```

Cross compiler

There is a draft concept of cross compilers in Forth published as `XCapp5.pdf` and `XCext5.pdf`.

There is no experience with cross compiler setups using recognizers yet. It may be worth to explore the possibilities in this area. The proposed scope concept fits nicely into the recognizer concept by e.g. switching the recognizer action modules (the `R:*` actions), but not the parsing actions (`REC:*`).

Not-Found Hooks

Many systems have a not-found hook that is called if a word is not found and is not a number. This hook is usually a deferred word. With recognizers it can be implemented as follows:

```
 : throw-13  -13 throw ;

 Defer interpret-notfound ( addr u -- )
   ' throw-13 is interpret-notfound
 Defer compiler-notfound ( addr u -- )
   ' throw-13 is compiler-notfound
 Defer postpone-notfound ( addr u -- )
   ' throw-13 is postpone-notfound

' interpret-notfound
' compiler-notfound
' postpone-notfound
recognizer: rec:notfound

' rec:notfound get-recognizers 1+ set-recognizers
```

With that the final action if a word could not be handled is a set of words that can be changed independently of the recognizer stack. These hooks are most useful for existing code that uses the not-found deferred word API. (Idea and basic code structure taken from gforth).