Forth Recognizer -- Request For Discussion

Author: Matthias Trute
Contact: mtrute@web.de
Version: 4
Date: May, 15 2017
Status: Draft

Change history

- 2014-10-03 Version 1 - initial version.
- 2015-05-17 Version 2 - extend rationale, added ' and [']
- 2015-12-01 Version 3 - separate use cases, minor changes for nested recognizer stacks. New `POSTPONE` action.
- **2016-09-03 Version 4 - Clarifications, Fixing typos, added test cases**
    - 2016-09-18 Added more test cases
    - 2016-09-25 Clarify that `>IN` is unchanged for an `R:FAIL` (`DT:NULL`) result.
    - 2016-10-21 simpler reference implementation
    - 2016-11-05 renamed keywords and concept names: data (type) token DT.
    - 2017-05-15 discussion of `LOCATE`

Background

I'm working on a Forth for 8-bit micro-controllers for more than 10 years now (amforth.sf.net). It is a useful tool for serious work and at the same time a nice playground for Forth too.

In 2011 my Forth got a floating point library. Since a micro-controller is (was) a ressource 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. 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 the running system to make it work 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 that implement recognizers. A third implementation is in the proposal.

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.

Solution

The monolithic design of the Forth interpreter is factored into three major blocks: First the interpreter. It maintains `STATE` and organizes the work. Second the actual data parsing. It is called from the interpreter and analyses strings (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 to the interpreter with 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.

Whenever the Forth text interpreter is mentioned, the standard words `EVALUATE` (CORE), `'` (tick, CORE), `INCLUDE-FILE` (FILE), `INCLUDED``(FILE), ``LOAD` (BLOCK) and `THRU` (BLOCK) are expected to act likewise. This proposal is not about to change these words, but to provide the tools to do so. As long as the standard feature set is used, a complete replacement with recognizers is possible.

The proposal is about the building blocks.

Proposal

XY. The optional Recognizer word set

XY.1 Introduction

The recognizer concept consists of two elements: parsing words that return data tokens 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 data tokens. A particular data token can be used by different parsing words.

A system provided data token is called `DT:NULL`. It is used if no other token is applicable. This token is associated with the system error actions if used in step e) of the text interpreter (see Appendix). It is used to achieve the action d) of the section 3.4 text interpreter.

A parsing word within the recognizer concept has the stack effect

```
REC:DATATYPE ( addr len -- i*x DT:TYPE | DT:NULL )
```

The parsing word must not change the string. Since it is called from the interpreter, it may access and, if successful, change `>IN`. If `>IN` is not involved, any string may serve as input, otherwise "addr/len" is assumed to be a substring of the buffer `SOURCE` points to.

"i*x" is the result of the parse action of the string "addr/len". `DT:TYPE` is the data token that the interpreter uses to execute the interpret, compile or postpone actions for the data `i*x`.

All three actions are called with the "i*x" data as left by the parsing word and are generally expected to consume it. They can have additional stack effects, depending on what `DT:TYPE:METHOD` actually does.

```
DT:TYPE:METHOD ( ... i*x -- j*y )
```

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

The names like `DT:TYPE`, `REC:DATATYPE` and `DT:TYPE:METHOD` don't have to actually exist, except the `DT:NULL` name.

A Forth system shall provide recognizers for integer numbers (both single and double precision) and the word look-up in the dictionary.

A recognizer stack is identified by its ID. The numeric value is system dependent and generally opaque. The elements of a recognizer stack are available with `GET/SET-RECOGNIZERS` only.

XY.2 Additional terms and notations

Data token

A cell sized number. It identifies the data type and a method set to perform the data processing in the text interpreter. The actual numeric value is system specific.

Recognizer

A string parsing word that returns data tokens together with the parsed data if successful. The string parsing word is assumed to run within the Forth interpreter and can access `SOURCE` and `>IN`.

Recognizer Stack

A stack of execution tokens of recognizers. The stack is identified with its stack id. This is a cell sized numeric value. It's numeric value is system specific.

XY.3 Additional usage requirements

XY.3.1 Data types

A data type token is a single cell value that identifies a certain data type. Append table the following table to table 3.1

Symbol	Data type	Size on Stack
dt	data type token	1 cell

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

- Change of the content of the parsed string during parsing.
- During `SET-RECOGNIZERS` the Recognizer stack size is exceeded. In this case, at least the following actions are possible
    - resize the stack, keeping its id
    - throw an exception
    - execute an error action

XY.4.2 Program documentation

No additional dependencies.

XY.5 Compliance and labeling

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

DT>COMP (DT:TYPE -- XT-COMPILE) RECOGNIZER

Return the execution token for the compilation action from the recognizer information token.

DT>INT (DT:TYPE -- XT-INTERPRET) RECOGNIZER

Return the execution token for the interpretation action from the recognizer information token.

DT>POST (DT:TYPE -- XT-POSTPONE) RECOGNIZER

Return the execution token for the postpone action from the recognizer information token.

DT:NULL (-- DT:NULL) RECOGNIZER

The null data token. It is to be used if no other data token can be used but a data token is needed. Its associated methods perform system specific error actions. The actual numeric value is system dependent.

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

Skip leading space delimiters. Parse name delimited by a space. Create a data token "name" with the three execution tokens.

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.

Each of the words XT-INTERPRET, XT-COMPILE and XT-POSTPONE represent has the stack effect `( ... i*x -- j*y )`. The words to compile and postpone the data shall consume the data "i*x". If the data "i*x" is on different locations (e.g. floating point numbers), these words shall use that data.

FORTH-RECOGNIZER (-- stack-id) RECOGNIZER

A system VALUE with a recognizer stack id.

It is `VALUE` that can be changed with `TO` to assign a new recognizer stack. This change has immediate effect.

The recognizer stack from this stack-id shall be used in all system level words like `EVALUATE`, `LOAD` etc.

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

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

The recognizer stack is unchanged.

RECOGNIZE (addr len stack-id -- i*x DT:TYPE | DT:NULL) RECOGNIZER

Apply the string at "addr/len" to the elements of the recognizer stack identified by `stack-id`. Terminate the iteration if either one parsing word returns a data type token that is different from `DT:NULL` or the stack is exhausted. In this case return `DT:NULL`.

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

REC-STACK (size -- stack-id) RECOGNIZER

Create a new recognizer stack with size elements.

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

Set the recognizer stack identified by `stack-id` with a new set of parsing words identified by the execution tokens `rec-n .. rec-1`. `rec-1` will be the parsing word that is called first, `rec-n` will be the last one.

If the size of the existing recognizer stack is too small to hold all new elements, an ambiguous situation arises.

XY.7 Reference Implementation

The code has as little as possible dependencies.

```
: REC-STACK ( size -- stack-id )
    1+ ( size ) CELLS HERE SWAP ALLOT
    0 OVER ! \ empty stack
;
\ create the default recognizer stack
4 REC-STACK VALUE FORTH-RECOGNIZER
```

```
: SET-RECOGNIZERS ( rec-n .. rec-1 n stack-id -- )
  2DUP ! CELL+ SWAP CELLS BOUNDS
  ?DO I ! CELL +LOOP
;

: GET-RECOGNIZERS ( stack-id -- rec-n .. rec-1 n )
   DUP @ >R R@ CELLS + R@ BEGIN
     ?DUP
   WHILE
     1- OVER @ ROT CELL - ROT
   REPEAT
   DROP R>
;

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

\ decode the data structure created by DT-TOKEN:
: DT>POST ( DT:TOKEN -- XT-POSTPONE ) CELL+ CELL+ @ ;
: DT>COMP ( DT:TOKEN -- XT-COMPILE  )       CELL+ @ ;
: DT>INT  ( DT:TOKEN -- XT-INTERPRET)             @ ;

\ the null token
:NONAME -1 ABORT" FAILED" ; DUP DUP DT-TOKEN: DT:NULL

: RECOGNIZE   ( addr len stack-id -- i*x DT:TYPE | DT:NULL )
    DUP >R @
    BEGIN
      DUP
    WHILE
      DUP CELLS R@ + @
      2OVER 2>R SWAP 1- >R
      EXECUTE DUP DT:NULL <> IF
        2R> 2DROP 2R> 2DROP EXIT
      THEN
      DROP R> 2R> ROT
    REPEAT
    DROP 2DROP R> DROP DT:NULL
;
```

A.XY Informal Appendix

A.XY.1 POSTPONE

`POSTPONE` compiles the data returned by `RECOGNIZE` (`i*x`) into the dictionary as literal(s) and appends the compilation action of the `DT:TOKEN` data token. Later at run-time the `i*x` data is read back and the compilation action is performed like it would have been called directly at compile time.

```
: POSTPONE ( "name" -- )
  PARSE-NAME FORTH-RECOGNIZER RECOGNIZE DUP >R
  DT>POST EXECUTE R> DT>COMP COMPILE, ;
```

This implementation assumes a system that uses recognizers only.

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 a wrapper for FIND to use addr/len as input
256 BUFFER: find-word-buf \ counted string
: 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 ;

:NONAME ( i*x XT +/-1 -- j*y )  \ INTERPRET
  DROP EXECUTE ;
:NONAME ( XT +/-1 -- )          \ COMPILE
  0> IF COMPILE, ELSE EXECUTE THEN ;
:NONAME ( XT +/-1 -- )          \ POSTPONE
  POSTPONE 2LITERAL ;
DT-TOKEN: DT:XT

: REC:FIND ( addr len -- XT +/-1 DT:XT | DT:NULL )
   find-word ( addr len -- XT +/-1 | 0 )
   ?DUP IF DT:XT ELSE DT:NULL THEN
;
```

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 (yet) part of the Forth standard, thus the postpone action here prints the number and throws an exception to enforce an error handling.

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

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

A.XY.3 Text Interpreter

The Forth text interpreter can be changed 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. The example is a full replacement if all necessary recognizers are available.

The algorithm of the Forth text interpreter as described in section 3.4 is modified. All subsections of 3.4 apply unchanged. Change the steps b) and c) from section 3.4 to make them optional, they can be performed with recognizers. Replace the step d) with the following steps d) to f)

d. For each element of the recognizer stack provided by `FORTH-RECOGNIZER`, 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 analyzed sub-string if successful. Otherwise it returns the system provided failure token DT:NULL 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 DT:NULL.

e. **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.

f. Continue with a)

```
: INTERPRET
  BEGIN
      PARSE-NAME DUP
  WHILE
      FORTH-RECOGNIZER RECOGNIZE
      STATE @ IF DT>COMP ELSE DT>INT THEN \ get the right XT from DT:*
      EXECUTE \ do the action.
      ?STACK  \ simple housekeeping
  REPEAT 2DROP
;
```

A.XY.4 Naming Conventions

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
DT:NUM	(-- n DT:NUM)	single cell numbers, based on >NUMBER
DT:DNUM	(-- d DT:DNUM)	double cell numbers, based on >NUMBER
DT:FLOAT	(-- DT:FLOAT) (F: -- f |)	floating point numbers, based on >FLOAT
DT:XT	(-- XT +/-1 DT:XT)	words from the dictionary, FIND
DT:NT	(-- NT DT:NT)	words from the dictionary, with name tokens NT

The matching recognizing parser words should be available as

Name	Stack effect
REC:NUM	(addr len -- n DT:NUM | d DT:DNUM | DT:NULL)
REC:FLOAT	(addr len -- DT:FLOAT | DT:NULL) (F: -- f |)
REC:FIND	(addr len -- XT +/-1 DT:XT | DT:NULL)
REC:NAME	(addr len -- NT DT:NT | DT:NULL)

The acronym DT stands for both *data type* and *data token*.

A.XY.5 Test Cases

```
T{ 4 REC-STACK constant RS -> }T

T{ :NONAME 1 ;  :NONAME 2 ;  :NONAME 3  ; DT-TOKEN: dt:1 -> }T
T{ :NONAME 10 ; :NONAME 20 ; :NONAME 30 ; DT-TOKEN: dt:2 -> }T

T{ : rec:1 NIP 1 = IF dt:1 ELSE DT:NULL THEN ; -> }T
T{ : rec:2 NIP 2 = IF dr:2 ELSE DT:NULL THEN ; -> }T

T{ r:1 DT>INT EXECUTE  -> 1 }T
T{ r:1 DT>COMP EXECUTE -> 2 }T
T{ r:1 DT>POST EXECUTE -> 3 }T

\ set and get methods
T{ 0 RS SET-RECOGNIZERS -> }T
T{ RS GET-RECOGNIZERS -> 0 }T

T{ ' rec:1 1 RS SET-RECOGNIZERS -> }T
T{ RS GET-RECOGNIZERS -> ' rec:1 1 }T

T{ ' rec:1 ' rec:2 2 RS SET-RECOGNIZERS -> }T
T{ RS GET-RECOGNIZERS -> ' rec:1 ' rec:2 2 }T

\ testing RECOGNIZES
T{         0 RS SET-RECOGNIZERS -> }T
T{ S" 1"     RS RECOGNIZE   -> DT:NULL }T
T{ ' rec:1 1 RS SET-RECOGNIZERS -> }T
T{ S" 1"     RS RECOGNIZE   -> DT:1 }T
T{ S" 10"    RS RECOGNIZE   -> DT:NULL }T
T{ ' rec:2 ' rec:1 2 RS SET-RECOGNIZERS -> }T
T{ S" 10"    RS RECOGNIZE   -> DT:2 }T
```

The dictionary lookup has the following test cases

```
T{ S" DUP" REC:FIND  -> ' DUP -1 DT:XT }T
T{ S" UNKOWN WORD" REC:FIND -> DT:NULL }T
```

The number recognizer has the following checks

```
VARIABLE OLD-BASE BASE @ OLD-BASE !

T{ S" 1234"   REC:NUM -> 1234 DT:NUM }T
T{ S" 1234."  REC:NUM -> 1234. DT:DNUM }T
T{ S" %-10010110" REC:NUM -> -150 DT:NUM }T
T{ S" %10010110"  REC:NUM ->  150 DT:NUM }T
T{ S" 'Z'"    REC:NUM -> char Z DT:NUM }T
T{ S" ABCXYZ" REC:NUM -> DT:NULL }T

\ check whether BASE is unchanged
T{ BASE @ OLD-BASE @ = -> -1 }T
```

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.
- FICL, a system close to Forth, has parse-steps since approx 2001.

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 `DT:<name>` for the data token word created with `DT-TOKEN:` for the data type "name".

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

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

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.

Test cases

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

```
T{ : S-1234 S" 1234" ; -> }T
T{ : D-1234 S" 1234." ; -> }T
T{ : S-UNKNOWN S" unknown word" ; -> }T
T{ : S-DUP   S" DUP" ; -> }T

T{ S-1234 FORTH-RECOGNIZER RECOGNIZE -> 1234  DT:NUM   }T
T{ D-1234 FORTH-RECOGNIZER RECOGNIZE -> 1234. DT:DNUM  }T
T{ S-DUP  FORTH-RECOGNIZER RECOGNIZE -> ' DUP -1 DT:XT }T
T{ S-UNKNOWN FORTH-RECOGNIZER RECOGNIZE  -> DT:NULL }T
```

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

```
T{ S-DUP     REC:FIND -> ' DUP -1 DT:XT }T
T{ S-UNKNOWN REC:FIND -> DT:NULL }T
T{ S-1234    REC:FIND -> DT:NULL }T
T{ D-1234    REC:FIND -> DT:NULL }T
```

```
T{ S-UNKNOWN REC:NUM -> DT:NULL }T
T{ S-1234    REC:NUM -> 1234 DT:NUM }T
T{ D-1234    REC:NUM -> 1234. DT:DNUM }T
T{ S-DUP     REC:NUM -> DT:NULL }T
```

If there are recognizers for size specific single and double cell numbers, they shall work as follows. The names are system dependent

```
T{ S-1234 REC:SNUM  -> 1234  DT:NUM   }T
T{ S-1234 REC:DNUM -> DT:NULL }T
T{ D-1234 REC:SNUM -> DT:NULL }T
T{ D-1234 REC:DNUM -> 1234. DT:DNUM }T
```

Floating point numbers are handled likewise

```
T{ : S-1234e5 S" 1234e5" ; -> }T
T{ S-1234e5 REC:FLOAT -> 1234e5 DT:FLOAT }
T{ S-1234e5 FORTH-RECOGNIZER RECOGNIZE -> 1234e5 DT:FLOAT }T
```

Discussion / Rationale

Some parts of the following text use the word names of the previous RFD revisions like `R:FAIL`. Despite they still apply.

Data Type Tokens

Earlier revisions of this RFD called them information tokens. In fact they describe a data type (e.g. a number) and they point to a set of action for the actions of the Forth core system: interpret, compile and postpone. These tokens are not related to the process they are created. They are only related to the data they are tied to hence the naming change. Esp. the `R:FAIL` is in fact not a failure but a null information, just like the 0 (zero) is the logical FALSE information.

`GET/SET-RECOGNIZERS`

These commands can create a deep data stack usage. They are modeled 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 `RECOGNIZE`, 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 change 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.

POSTPONE

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.

Bernd Paysan wrote in clf

> Concerning the postpone action and ' and ['] using recognizers: IMHO, there's not much point in generating a super-efficient postpone, but you can use ' and ['] together with literals, if the postpone method is modified to *only* contain the work to save the i*x part of the recognizer output into the dictionary. The remaining action of postpone is generic. So POSTPONE executes the literal-append part of the r:token and then appends the r:token as literal and the compilation part of the r:token.
>
> ' and ['] can check if the literal-append part is empty (a noop), and if not, create a quotation that contains that literal, and appends the dt>int part of the table. I.e. ['] 3 becomes something like [: 3 noop ;], with an easy opportunity to optimize away the noop.
>
> This is not mandatory, but I'd like to implement it that way. And that means the postpone part has to be changed to the essential core (the handling of the recognizer-specific i*x), and the rest is done by POSTPONE.
>
> That means dt:num is defined as
>
> ```
> : lit, postpone literal ;
> ' noop ' lit, ' lit, dt-token: r:num
> ```
>
> and POSTPONE is defined as
>
> ```
> : POSTPONE ( "name" -- )
>   PARSE-NAME FORTH-RECOGNIZER  RECOGNIZE  >R
>   R@ dt>post EXECUTE r> dt>comp COMPILE, ;
> ```
>
> following your reference implementations.
>
> This also makes the simple two-part table easier to implement, as *only* the compilation part (perform literal part+append interpretation part) needs to be generated.

From 6.1.2033 POSTPONE: "Append the compilation semantics of name to the current definition." This postpone does exactly this.

The suggested ' is part of the implementation and can be left to the system provider.

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. It has to communicate back, where it finished its work so that subsequent parse action start at the right point. There are a few possibilities

- The input for recognizer comes from within SOURCE and is managed with >IN. That is the designated environment for recognizers. Systems are free to make a copy of the word before calling the parsing words from the recognizer. A multi-word recognizer nevertheless needs access the SOURCE buffer and changes >IN accordingly. It must not change the content of the string however.

- The input comes from an arbitrary string. SOURCE and >IN are not used. The word RECOGNIZE has to tell now, how far it went in addition to the actual results. The standard already has a word that works that way: >NUMBER (ud1 addr len -- ud2 addr' len'). A similiar RECOGNIZE would have the stack effect (addr len -- i*x addr' len' DT:TOKEN | DT:NULL).

Since many standard words are already grouped around SOURCE and >IN it seems to be overkill to maximize the flexibility. That's why option 1 is preferred. Furthermore it leads to simpler code and easier integration into existing systems. There is no dependency on SOURCE and >IN for the single-word recognizer use case.

Another aspect with multiword recognizers is that it is possible that the closing syntactic element of the multi-word sentence is not within the current input string One or more REFILL may be necessary to get it. Since that may be troublesome in the long run, the closing element shall be in the same input string as the opening one.

Keep the Interpreter

The Forth 2015 meeting in Bath as well as (earlier) Andrew Haley added the wish / requirement to keep the current interpreter and make recognizers an truly optional part. Changed in the proposal to make the Forth 2012 interpreter steps to search the dictionary (step b) and convert numbers (step c) optional. That way the current interpreter can work without changes and at the same time the hard coded steps b) and c) from section 3.4 could be replaced with recognizers. The recognizer steps are added as step d) to f) It should be clear that the example implementation of the interpreter is not mandatory.

Nevertheless the full power of the concept cannot be achieved with such a two-class interpreter. For that, one need to be able to replace the standard actions FIND and number recognition too.

As a related change the words DT>COMP, DT>INT and DT>POST became part of the proposal since they are needed to write an interpreter and similar words portably.

Switching Recognizer Stacks

The Forth 2015 meeting wishes a possibility to switch between prepared recognizer stacks. To achieve this, the words SET-RECOGNIZERS, RECOGNIZE and GET-RECOGNIZERS are changed to have an additional parameter stack-id that identifies the recognizer stack to be used. The elements of the recognizer stack may not be accessible with the normal fetch and store operation, the numeric value of the stack-id is implementation defined. The stack may have a limited size too resulting in an error condition if the maximum size is exceeded.

The new word FORTH-RECOGNIZER is introduced to have a global (drift) anchor to provide a common starting point to be used by various words like EVALUATE from whom a consistent behavior is expected. It is a VALUE to switch the whole stack at once.

Nesting Recognizer Stacks

An extension of the Switching Recognizer Stacks.

Example is a number recognizer. Instead of checking for all number formats from the Forth 2012 spec in one recognizer, every variant is handled by an individual one. All number checks are collected in the recstack:numbers recognizer stack.

```
\ 'y'
: REC:CHAR ( addr len -- n DT:NUM | DT:NULL )
  ....
;
\ single cell numbers
: REC:SNUM ( addr len -- n DT:NUM | DT:NULL )
  ...
;
```

```
\ double cell numbers
: REC:DNUM ( addr len -- d DT:DNUM | DT:NULL )
  ...
;

3 RECOGNIZER CONSTANT recstack:numbers

' REC:CHAR ' REC:SNUM ' REC:DNUM 3 recstack:numbers SET-RECOGNIZERS

: REC:NUM ( addr len -- n DT:NUM | d DT:DNUM | DT:NULL )
  recstack:numbers RECOGNIZE
;

' REC:NUM ' REC:FIND 2 FORTH-RECOGNIZER SET-RECOGNIZERS
```

Flags, `DT:NULL` or Exceptions

The `DT:NULL` 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 `DT:*` token with the constant `DT:NULL` can be easily optimized.

A completely different approach is using exceptions to deliver the flag information from `RECOGNIZE` 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 sequence of compare operations. The third argument against exceptions is that if used for recognizers, they mandate too much implementation details on system providers which is not considered useful.

That `DT:NULL` is used in two ways is an optimization. The flag information can be carried with the equation `DT:* DT:NULL <>` as well.

No `REC:FAIL`

That there is no final `REC:FAIL` in the recognizer stack is an optimization 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 recognize this as an error and crashed without further notice. To circumvent this situation, the current recognizer stack depth is needed. 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 has to be kept, essentially making appending a recognizer always an insert-in-the-middle action.

`DT:TOKEN`

Every recognizer returns the data and a token, called `DT:TOKEN`. This token is used to identify a data type and it provides all information necessary to handle the data inside the interpreter. Each data item is used in three different actions: interpret, compile and postpone. The interpret and compile action are used depending on `STATE`. The postpone action serializes the data and adds the data specific compile action to be executed later on.

This design follows the name tokens and `TRAVERSE-WORDLIST` from the Programming Tools wordset.

LOCATE

LOCATE is an interactive tool found in many Forth systems to display information about an item <something> that follows immediately in the input line. LOCATE is non-standard and may thus has different meanings and implementations. It usually depends on carnal knowledge of the system.

With recognizers the fear came up, that a LOCATE may not work any longer due to complex syntactic schemas that are not easy to handle.

Existing Practice

Common usage is LOCATE word giving a brief information where the source code of the definition can be found or directly displaying this information.

Only words in wordlists are subject to be LOCATE'd. Numbers and other literal-like data are not expected to work and produce various error messages.

The actions taken during LOCATE can be customized in many ways, defers, macros and substitions are used.

Gforth (file *locate.fs*): LOCATE word opens a file called *TAGS*, searches there for word, constructs a command line from the information found to invoke the vi editor and executes it. If something unexpected happens exceptions are thrown at various stages.

Swiftforth has a header field for LOCATE information, VFXlin keeps somehow track of the file names during compilations. Both systems use them to display display the data and/or execute command lines.

Possible implementations

The first approach assumes that the information LOCATE uses are tied to the item itself. E.g. a header element in the wordlist entry. The systems that go that way have words that make header information available starting from the execution tokens (XT) or the name token (NT). This information is part of the usual RECOGNIZE step.

E.g. LOCATE FOO may display "UNKNOWN" assuming FOO is not defined anywhere. LOCATE IF may display "XT address 1" (for an immediate IF) a user supplied recognizer (for simplicity the name token lookup) may display "NT address". A -> recognizer that implements the TO operation can display the "TO address" information from the (hypothetical) DT:TO data token.

System specific knowledge in a DT>STRING transforms the DT:XT token into something readable. This is similar to the NAME>STRING. The recognizer stack that is used to identify the data may be the same as the text interpreter is supposed to use (FORTH-RECOGNIZER). That way the LOCATE can be implemented as

```
: LOCATE
  DEPTH N>R \ save current data
  PARSE-NAME FORTH-RECOGNIZER RECOGNIZE DISPLAY-DT-DATA
  NR> DROP   \ restore previously saved data
;
```

with DISPLAY-DT-DATA to show the data actually is something like

```
: DISPLAY-DT-DATA DT>STRING TYPE DEPTH 0 ?DO . LOOP ;
```

This DISPLAY-DT-DATA can be expanded to work with any system provided recognizer and may have a hook for user supplied ones.

The second version of LOCATE is a recognizer itself. This is illustrated for the TAGS file based LOCATE as in gforth. The recognizer returns a new data type token, called DT:TAGS This data type token does not need support compiling and postponing actions. The LOCATE command uses the interpret action only. The parsing action may be located in a recognizer stack of its own or may be added temporarily to the standard stack.

```
:NONAME ( addr len -- ) TYPE ; :NONAME 2DROP ; DUP DT-TOKEN: DT:TAGS
: REC:TAGS ( addr len -- addr' len' DT:TAGS | DT:NULL )
   \ open TAGS file, search for addr/len and create a new
   \ string with data from the TAGS file at addr' len' if found
;
1 REC-STACK LOCATE-RECOGNIZER
' REC:TAGS 1 LOCATE-RECOGNIZER SET-RECOGNIZERS
: LOCATE PARSE-NAME LOCATE-RECOGNIZER RECOGNIZE DT>INT EXECUTE ;
```

With the `LOCATE-RECOGNIZER` as a separate stack user supplied data type tokens can be added to the LOCATE stack easily. Moreover any non-locate-able strings (literals) are handled automatically without interfering with other data locations (floating point stack) due to the standard `DT:NULL` action.

`DT:NULL` necessity

Comparing the different implementations. Esp the dual use as a flag and a token is discussed with code examples.

Exceptions are not an option as already discussed.

Parse `REC:*` *actions*

For simplicity the recognizer for floating point numbers.

With `DT:NULL`

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

Without `DT:NULL`

```
: REC:FLOAT ( addr len -- ( DT:FLOAT | 0 ) ( F: -- f )
  >FLOAT IF DT:FLOAT ELSE 0 THEN ;
```

Conclusion: almost the same.

`RECOGNIZE`

with `DT:NULL`

```
: RECOGNIZE   ( addr len stack-id -- i*x DT:TOKEN | DT:NULL )
    DUP >R @
    BEGIN
      DUP
    WHILE
      DUP CELLS R@ + @
      2OVER 2>R SWAP 1- >R
      EXECUTE DUP DT:NULL <> IF
        2R> 2DROP 2R> 2DROP EXIT
      THEN DROP R> 2R> ROT
    REPEAT
    DROP 2DROP R> DROP DT:NULL
;
```

Without DT:NULL

```
: RECOGNIZE   ( addr len stack-id -- i*x DT:TOKEN | 0 )
    DUP >R @
    BEGIN
      DUP
    WHILE
      DUP CELLS R@ + @
      2OVER 2>R SWAP 1- >R
      EXECUTE DUP IF
        2R> DROP 2R> 2DROP EXIT
      THEN DROP R> 2R> ROT
    REPEAT
    DROP 2DROP R> DROP DT:NULL
;
```

again, almost the same.

POSTPONE

with DT:NULL

```
: POSTPONE ( "name" -- )
  PARSE-NAME FORTH-RECOGNIZER RECOGNIZE DUP >R
  R>POST EXECUTE R> R>COMP COMPILE, ;
```

without DT:NULL

```
: POSTPONE ( "name" -- )
  PARSE-NAME FORTH-RECOGNIZER RECOGNIZE
  ?DUP IF
    DUP >R
    DT>POST EXECUTE R> DT>COMP COMPILE,
  ELSE
    NOT-RECOGNIZED
  THEN ;
```

special casing "not-recognized" and slightly more complex due to NOT-RECOGNIZED.

Interpreter

With DT:NULL

```
: INTERPRET
  BEGIN
      PARSE-NAME DUP
  WHILE
      FORTH-RECOGNIZER RECOGNIZE
      STATE @ IF DT>COMP ELSE DT>INT THEN \ get the right XT from DT:*
      EXECUTE \ do the action.
      ?STACK  \ simple housekeeping
  REPEAT 2DROP
;
```

Without DT:NULL

```
: INTERPRET
  BEGIN
      PARSE-NAME DUP
  WHILE
      FORTH-RECOGNIZER RECOGNIZE
      ?DUP IF \ we got an DT:* table
        STATE @ IF R>COMP ELSE R>INT THEN \ get the right XT from DT:*
        EXECUTE \ do the action.
      ELSE
         \ no recognizer did the job
         NOT-RECOGNIZED
      THEN
      ?STACK  \ simple housekeeping
  REPEAT 2DROP
;
```

Like `POSTPONE` special casing the "not-found" condition and slightly more complex due to `NOT-RECOGNIZED`.

Adapting the special case "not recognized" requires extending the text interpreter specification too.

Conclusion

`DT:NULL` is essential since it simplifies both the concept and the implementation by not special casing any result. Furthermore the code for the recognizers is easier to read and understand: `DT:NULL` vs `0`. "Notation matters".

Use Cases

There are a number of use cases that benefit from or use language features. These use cases are purely informative.

Name Tokens

Name Tokens (nt) are part of the Forth 2012 Programming Tools word set. This section is just a use case description deploying an optional 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 DT:NAME|DT:NULL)`. 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 POSTPONE LITERAL     ;   ( nt -- ) \ postpone
DT-TOKEN: DT:NAME
```

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

```
\ the analogon to search-wordlist
: search-name ( addr len wid -- nt | 0 )
  >R 0 \ used as flag inside the following quotation
  [: ( addr len flag nt -- addr len false true | nt false )
    >R DROP 2DUP R@ NAME>STRING COMPARE
    IF R> DROP 0 -1 ELSE 2DROP R> 0 THEN
  ;] R> TRAVERSE-WORDLIST ( -- addr len false | nt )
  DUP 0= IF NIP NIP THEN
;

\ a single wordlist is checked
: (rec:name)    ( addr len wid -- nt DT:NAME | DT:NULL )
  search-name
  ?DUP IF DT:NAME ELSE DT:NULL THEN
;

\ checks only the standard word-list
: rec:name ( addr len -- nt DT:NAME | DT:NULL )
  FORTH-WORDLIST (rec:name)
;
```

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.

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 modes of the interpreter
: (interpret-i) DT>INT EXECUTE ;
: (interpret-c) DT>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
```

```
      FORTH-RECOGNIZER RECOGNIZE  \ analyze it
      (interpret)     \ act on it
      ?stack          \ simple housekeeping
    REPEAT 2DROP
 ;
```

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
DT-TOKEN: DT:notfound

: rec:notfound ( addr len -- )
   DT:notfound
;
```

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

`'` and `[']`

`'` (tick) and its companion `[']` (bracket-tick) are affected too. It is common practice that the sequence `' foo execute` does the same as calling `foo` directly (in interpret mode). Now consider 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 recognized item is an executable "tick-able" word. E.g. numbers and compile-only words are not.

Implementation requires system specific knowledge. The following code depends on `DT:XT` to work.

```
: executable? ( DT:TOKEN -- f )
         DT>INT \ get the interpretation action for the given token
   DT:XT DT>INT \ get the system specific interpret action
   =
;

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

Alternative `DT>x` Stack Effect

in gforth, the `DT>x` words have a different stack effect. They use not only the DT token information but the whole data set as returned from the parsing words. The returned data may modify this data too. In the reference implementation this is not used however.

DT>COMP (i*x DT:TOKEN -- j*y XT)

The primary purpose of this seems to be an optimization and an unification of the name token and the data type token for named words), so that DT>COMP becomes identical to NAME>COMPILE for name tokens.

Older Remarks

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 `RECOGNIZE` action (`REC:*` words) and the literal-like and final-action set replaces the proposed 3 method set in `DT:*`. 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: DT: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.

Acknowledgments

The following people did major or minor contributions, in no particular order.

- Bernd Paysan
- Jenny Brien
- Andrew Haley
- Alex McDonald
- Anton Ertl