# The Grand Recognizer Unification

Bernd Paysan
net2o

M. Anton Ertl*
TU Wien

## Abstract

There is an obvious similarity between the search order and a recognizer sequence, which has led to similarities in proposed words (e.g., `get-recognizer` is modeled on `get-order`). By turning word lists into recognizers, we unify these concepts. We also turn recognizer sequences (and be extension the search order) into a recognizer, which allows nestable recognizer sequences and wordlist sequences in the search order. The implementation becomes simpler, too.

## 1 Introduction

Data uniformity is a useful principle in programming, because it means that we can use the same program on more different data. Examples of uniformity are Unix's principles of 1) accessing many different kinds of things (files, devices, pipes) as file (using open, close, read and write) and 2) to organize files as sequences of bytes. Object-oriented programming allows to treat, e.g., circles and triangles uniformly as graphical objects, with higher-level code being able to e.g., draw graphical objects.

In the Forth world, examples of uniformity are words encompassing colon definitions, constants, variables, etc., and cells encompassing addresses, signed and unsigned integers.

Different types of data can be usefully unified if they have commonalities. In this paper we unify four things: recognizers, wordlists, recognizer sequences, and the search order. Their commonality is that you pass a string to them, and they either recognize it (and produce some data representing the string), or not (and produce a *not-found* result); see Fig. 1.

Section 2 gives examples of how each of the three other concepts works as recognizer, how you can implement existing interfaces, and in some cases how you can implement the concept. Section 3 describes the implementation of wordlists as recognizers in Gforth. In Section 4 we discuss related work.

*anton@mips.complang.tuwien.ac.at

| | wordlist | recognizer |
|---|---|---|
| one | wordlist<br>`find-name-in` | recognizer<br>`execute` |
| many | search order<br>`find-name` | recognizer sequence<br>`recognize` |

Figure 1: Similar concepts and searching words for recognizing a string

## 2 Grand Unified Recognizers

### 2.1 Wordlists as recognizers

We start by generalizing wordlists to also be recognizers: The *wid* is implemented as the xt of a `rec-nt`-like[1] recognizer that recognizes the words in the wordlist. Note that this requires changes pretty deep in the the Forth system (see Section 3), that's why we do not propose this change for standardization. The resulting wordlist recognizers have the stack effect

```
( c-addr u -- nt rectype-nt | rectype-null)
```

The benefit is that we can now use wordlists wherever recognizers are expected, e.g., in recognizer sequences (see Section 2.3).

But first, how can we use it as a wordlist? You can implement `find-name-in` as follows:

```
: find-name-in ( c-addr u wid -- nt | 0 )
   execute rectype-nt <> if 0 then ;
```

### 2.2 Recognizer sequences as recognizers

This part has been proposed[2] as a potential part of the recognizer proposal.

It is straightforward to implement a recognizer sequence as recognizer:

[1] https://forth-standard.org/proposals/recognizer
[2] https://forth-standard.org/proposals/nestable-recognizer-sequences

```
: rec-sequence ( xt1 .. xtn n "name" -- )
  create dup , dup , 0 ?do , loop
does> ( c-addr u -- ... rectype )
  {: c-addr u addr :}
  addr cell+ @ cells addr 2 cells +
  dup >r + r> ?do
    c-addr u i @ execute
    dup rectype-null <> if
      unloop exit then
  1 cells +loop
  rectype-null ;
```

This implementation stores the maximum size in the first cell, and the current size in the second cell, followed by the recognizers in the sequence.

The benefit of having recognizer sequences as recognizers is that we can, e.g., put it in another recognizer sequence, i.e., recognizer sequences become nestable. So even if each sequence is short, a sequence can contain an unlimited number of basic recognizers.

We have the following accessor words for recognizer sequences:

```
\ we expect the following words
\ is-defer? ( xt -- f )
\ is-rec-sequence? ( xt -- f )

\ helper word
: follow-defers ( xt1 -- xt2 )
  begin
    dup is-defer? while
      defer@
  repeat  ;

: get-rec-sequence ( xt -- xt1 .. xtn n )
  follow-defers dup is-rec-sequence? 0= if
    drop 0 exit then
  >body cell+ dup cell+ over @
  dup >r cells rot + ?do
    i @ -1 cells +loop
  r> ;

: set-rec-sequence ( xt1 .. xtu u xt -- )
  follow-defers
  dup is-rec-sequence? 0= -12 and throw
  >body {: u addr :}
  u addr @ > -49 and throw
  u addr cell+ !
  addr 2 cells + dup u cells + rot ?do
    i !
  1 cells +loop ;
```

## 2.3 Search order as recognizer

With wordlists as recognizers, we can implement the search order as a recognizer sequence:

```
: rec-nothing ( c-addr u -- rectype-null )
  \ recognizer that recognizes nothing
  2drop rectype-null ;

' rec-nothing dup 2dup 2dup 2dup 2dup
  2dup 2dup 2dup 16 rec-sequence rec-nt0

: get-order ( -- wid1 ... widu u )
  ['] rec-nt0 get-rec-sequence ;

wordlist constant root-wordlist

: only ( -- )
  root-wordlist 1 rec-nt0 set-rec-sequence ;

: set-order ( wid1 ... widn n -- )
  dup -1 = if drop only exit then
  ['] rec-nt0 set-rec-sequence ;
```

You can put non-wordlist recognizers in the search order, if they have the stack effect

```
( c-addr u -- nt rectype-nt | rectype-null)
```

In particular, you can put recognizer sequences containing wordlists in the search order. Putting other kinds recognizers in the search order will result in `find-name` not working properly.

### Locals

One issue in implementing standard Forth is how to find locals. The standard does not really specify the details, but the text interpreter certainly has to find them when they are in scope, they probably should not be in the search order, and whether `find-name` (or `find`) should find them is not entirely clear and has been answered differently by different systems.[3]

I think that `find-name` should find locals, and here I show how that can be done. I assume that there is a system-specific `rec-loc` that recognizes locals and behaves like `rec-nt` (it can be implemented as a temporary wordlist, or with a separate mechanism). First, we implement `rec-nt`:

```
defer rec-nt

' rec-nt0 ' rec-loc 2 rec-sequence rec-locals

: activate-locals ( -- )
  ['] rec-locals is rec-nt ;
: deactivate-locals ( -- )
  ['] rec-nt0 is rec-nt ;

deactivate-locals
```

With that, `find-name` is easy to implement:

[3] https://forth-standard.org/standard/locals#reply-426

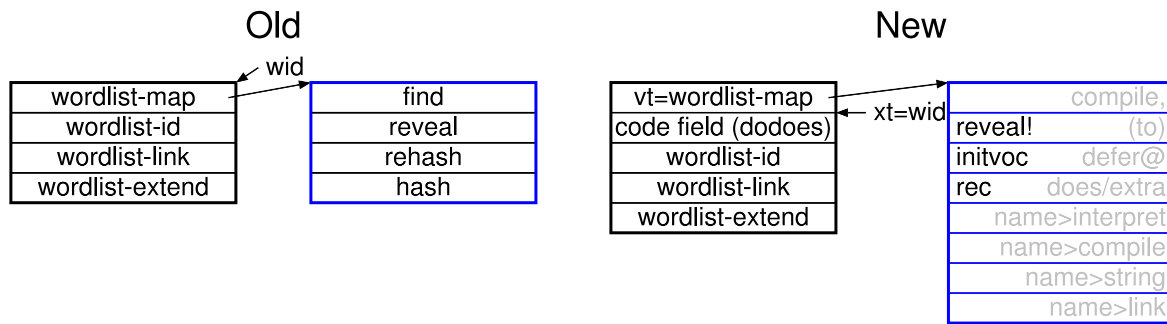


Figure 2: Wordlist implementations in Gforth

```
: find-name ( c-addr u -- nt|0 )
  ['] rec-nt find-name-in ;
```

# 3 Wordlist Implementation

The previous section shows possible implementations of recognizer sequences and the search order. This section discusses the implementation of wordlists.

Figure 2 shows two implementations of a wordlist in Gforth. We look at the old implementation first, because the new implementation is based on it.

## 3.1 Old wordlist implementation

In the old implementation a wordlist is a structure with the following fields:

**wordlist-map** This field points to a virtual method table (map) of methods for the wordlist; the methods will be explained below.

**wordlist-id** This (badly named) field contains the start of the linked-list representation of the wordlist. In hashed wordlists the linked-list representation is not used for searching (for performance reasons), but it is used for `words` and `traverse-wordlist`. For hashed wordlists it is also used as persistent representation of the wordlist, from which the hashed representation can be recreated whenever it is convenient.

**wordlist-link** This field points to the next wordlist in the linked list of wordlists (`voclink`), used in various places where all wordlists or all words are needed.

**wordlist-extend** This field may contain additional data. In hashed wordlists it contains a unique number for the wordlist, which is used to ensure that two words with the same name, but in different wordlists are sorted into different hash table buckets.

The method fields in the wordlist-map contain the xts of the method implementations. In the old implementation, methods were invoked with sequences like

```
dup wordlist-map @ reveal-method perform
```

instead of with separate words. The method fields are:

**find-method** `( addr len wid - nt|0 )` `find-name-in` for the wordlist; E.g., for a normal `wordlist`, this implements a case-insensitive hashed lookup.

**reveal-method** `( nt wid - )` inserts nt into the wordlist.

**rehash-method** `( wid - )`

**hash-method** `( wid - )` Both the hash and the rehash methods are used nowadays for putting all the entries of a hashed wordlist into the hash table (do nothing for a linked-list wordlist). There used to be a difference between the methods in earlier times.

## 3.2 New wordlist implementation

The new wordlist implementation is a nameless recognizer word (cf. Figure 3 of [PE19]). The fields are essentially the same as in the old implementation: `wordlist-id`, `wordlist-link` and `wordlist-extend` are exactly the same; they reside in the parameter field of the nameless word.

A new field is added: the code field of the word; the wordlist recognizers are implemented as `create...does>` words, so the code field contains `dodoes`[4].

Both wordlists and (with the new Gforth header [PE19]) words have a virtual method table, pointed to by `wordlist-map` in wordlists, and by vt in word headers. We chose to use the vt as `wordlist-map`.

[4] `Dodoes` is the code address of a piece of native code that pushes the parameter field address and then executes the xt in the does/extra field

```
find-method ( addr len wid -- nt|0 )
```

is replaced by

```
rec-method ( addr len wid --
      nt rectype-nt | rectype-null )
```

This rec-method must be invoked by `execute`ing the wid, so the does/extra method serves as the rec-method.

The other methods now have invocation words: `reveal!` for `reveal-method`, and `initwl` for what used to be `rehash-method`/`hash-method`[5].

For the `reveal!` and `initwl` methods there are no natural places in the word header. We reused the `(to)` (aka `defer!`) entry for `reveal!` and the `defer@` entry for `initwl`. This means that users who apply `defer@` and `defer!` to a wid will not get an error message, but some other behaviour that is probably unexpected; but then, it is typical of Forth's approach to type-checking that the caller of a word only passes parameters of the right type (i.e., only the xts of deferred words to `defer@`), and gets arbitrary behaviour otherwise.

Alternative approaches would have been to 1) append additional entries for the `reveal!` and `initwl` methods to the virtual method table; or to 2) have a wordlist-map field separate from the vt, which points to a method table containing the `reveal!` and `initwl` method xts.

Implementing wordlists as Gforth words also offers other options that we have not implemented (yet): `Name>string` could return the name of the vocabulary or the constant associated with the wordlist (if there is one), making it easier to implement, e.g., `order`. `Name>link` could follow `wordlist-link`, allowing to implement `voclink` (the list of wordlists) as a wordlist itself.

# 4 Related work

Matthias Trute developed the idea of recognizers into a real, workable implementation [Tru11] and proposal for standardization [Tru15].

Ertl [Ert15, Section 4.1, 5.1] has presented other unifying approaches: A variant of recognizers that generate temporary (or permanent) words, with `find-name` to find them. The paper also discusses the disadvantages of these approaches, and these have eventually led to deciding against these and for the RfD approach.

Ertl mentioned the idea of a "virtual wordlist consisting of a changeable search order of sub-wordlists" in 2003[6].

[5] There has been no difference between these methods in recent years, so they have been combined into one; also, we have not given this method a field name.

[6] <2003Apr18.141759@a0.complang.tuwien.ac.at>

# 5 Conclusion

By unifying recognizers, wordlists, recognizer sequences, and the search order we can write words which can deal with all these things. The benefits already start when implementing these concepts: the search order becomes just another recognizer sequence, simplifying the implementation of `find-name`, `get-order` and `set-order`.

# References

[Ert15] M. Anton Ertl. Recognizers — why and how. In *31st EuroForth Conference*, pages 77–78, 2015. 4

[PE19] Bernd Paysan and M. Anton Ertl. The new Gforth header. In *35th EuroForth Conference*, pages 5–20, 2019. 3.2

[Tru11] Matthias Trute. Recognizer — interpreter dynamisch verändern. *Vierte Dimension*, 27(2):14–16, 2011. 4

[Tru15] Matthias Trute. Forth recognizer — request for discussion. 3rd RfD, Forth200x, 2015. 4