Recognizers: Arguments and Design Decisions

M. Anton Ertl*
TU Wien

Abstract

The Forth text interpreter processes words and numbers. Currently the set of words can be extended by programmers, but not the recognized numbers. User-defined recognizers allow to extend the number-recognizer part, too. This paper shows the benefits of recognizers and discusses counterarguments. It also discusses several design decisions: Whether to define temporary words, or a set of interpretation, compilation, and postponing actions; and whether to hook the recognizers inside `find` or in the text interpreter.

1 Introduction

A strength of Forth is its extensibility. You can define new words to build an application-specific language, and then program in that language (or at least that's a frequently-told tale). However, the text interpreter conists of two parts: dealing with dictionary words and dealing with numbers; and while the former is extensible, the latter is not (in standard programs).

A recognizer tries to recognize a class of strings (e.g., numbers), and, if successful, provides the necessary information for text-interpreting it in the recognized sense; e.g., push the value of the number during interpretation or (for compilation) at runtime.

In this paper, we first look at the benefits of introducing recognizers (Section 2), then discuss some counterarguments (Section 3. We also look at two design decisions: Whether to let the recognizers define temporary words or or a set of interpretation, compilation, and postponing actions (Section 4), and where recognizers should hook in (Section 5). Finally, we look at the history of recognizers (Section 7).

2 Benefits and Uses

This section describes some benefits of implementing recognizers, in particular some uses.

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

2.1 Factoring of numbers

Gforth's (integer) number parsing has been a horrible mess. A long time ago I tried to refactor it to be less horrible, but the result was not much better; I particularly dislike the words with variable stack effects due to the words handling both single-cell and double-cell numbers. Other Forth systems have similar horrors in this area. This failure is probably due to my shying away from refactoring the text interpreter itself.

Recognizers provide an easy way to solve the variable stack-effect problem: Have one recognizer for single-cell numbers and one for double-cell numbers. Of course this kind of factoring is also possible without support for user-defined recognizers, but the implementation difference to also supporting user-defined recognizers would be small.

That being said, the current implementation in Gforth (by Bernd Paysan) uses one number recognizer that calls the not-much-better-factored words, but now that is easy to change.

2.2 Floating-point numbers

Standard floating-point numbers require a recognizer for the FP numbers. Most Forth systems initially just support the (integer) number recognizer, and add the FP recognizer at a later point in time (sometimes only after user intervention); they typically use a hook for this that is used for this particular purpose. User-defined recognizers are a generalization of this principle.

2.3 Other literals

SwiftForth supports various non-standard ways to write doubles, such as 2016-09-07, 9/7/2016, 7.9.2016, which is supposedly good for writing dates or telephone numbers, but, as you can see from these examples, where we get three different doubles for the same date, the technique has significant limitations. A full-blown recognizer for dates (or three, one for each date syntax) will interpret the correct fields as year, month, and day, depening on the separator character, and also perform the conversion into an appropriate format.

2.4 Parsing words

In the absence of user-defined recognizers, people have written parsing words as a workaround, e.g.

```
char a   \ made unnecessary by 'a'
[char] a \ made unnecessary by 'a'
' word
['] word
s" some string"
```

Some people have even claimed that the parsing words are the Forth way to do such things, but the fact that Forth originally had a (non-extensible) number recognizer and no `s"` is counterevidence to this claim.

`s"` has the problem of surprising behaviour in some corner cases in many systems (e.g., due to `state`-smartness [Ert98] or because the replacement of `state`-smartness in some implementations does not cover all corner cases, either).

`'/[']` does not have corner-case problems, but it has the problem that the frequent case of cutting and pasting code between interpreted code and compiled code requires changing the code.

By adding a recognizer for `"some string"`, we get rid of the corner cases. One unusual thing here is that the recognized string can extend across a white space; while recognizers get a `parse-name`-parsed string as parameter, they can do their own parsing. However, that parsing is always done at text-interpret time, avoiding the problems of `state`-smartness etc.

Likewise, we can add a recognizer that recognizes `'word` and produces the xt of `word`, and it will work like `' word` interpretively, and like `['] word` while compiling.

2.5 `to`

Parsing words are not just used for literals; `to` is also a parsing word that has the same problems as `s"` (except that the corner cases are ambiguous conditions in the standard), and recognizers can also be used to replace `to`. Gforth has a recogizer that recognizes `->`*x*, and that is equivalent to `to` *x* (for locals, values etc.) and `is` *x* (for deferred words).

2.6 Dot-Notation Parser

One problem that Forth has had is the naming of structure and object fields. Frequent field names are `next`, `count`, `val`, `left`, `right`, but you normally don't want to define them more than once (and, in the case of `count`, the name is already taken).

One workaround has been to include the structure name in the name, e.g. `list-next`, but with inheritance of fields in object-oriented programming, this does not work so well: e.g., you would have `intlist-val` for the val field, but `intlist-next` would not be defined (instead, `list-next`). Some object-oriented systems (in particular, `objects.fs` [Ert97]) work around this problem by putting the fields in class-specific wordlists and changing the search order appropriately, but that restricts field access to only the current object (or at least the current class).

Therefore, a desired and missing feature in Forth has been to change the search order for one word only, in order to use the right field word (among a number of such words with the same name) without too much ado. One example of this desire is the Prelude concept [Mah98]; another is the dot-notation parser of ClassVfx OOP [MPE16, Section 29.11]. In the dot notation, if you have a type `Point` with field `x` and an instance `MyPoint` of type `Point`, you can access the field `x` either with `MyPoint.x` or with `MyPoint Point.x`.

A dot-notation parser can be implemented as a recognizer.

2.7 `Postpone, ', and [']`

In standard code, when you want to postpone a literal, you cannot do it directly, but have to find a workaround. E.g., write `5 postpone literal` instead of `postpone 5`; that's also true for literals produced by parsing words: instead of `postpone s" bla"`, you have to write `s" bla" postpone sliteral`.

And it's also true for other things you do with parsing words: when you want to postpone `to this`, you cannot do it directly, but have to define

```
: to-this to this ;
```

and then `postpone to-this`.[1]

Recognizers (as proposed in the RfD) support `postpone`ing recognized strings. One benefit is that this feature allows writing smaller and easier-to-read code, but the main benefit is that it closes the hole that made the workarounds necessary.

One recognizer approach (Section 4.1) also supports ticking recognized strings, so one could write `'15` instead of having to define

```
15 constant fifteen
```

first and then writing `'fifteen`.

3 Counterarguments

There have been quite a number of negative reactions to the proposal for user-defined recognizers. They are generally not technical, but nevertheless, let's examine some of the arguments.

[1] If you think that this is a contrived problem, you are wrong. This problem and this solution occur in `objects.fs` [Ert97].

3.1 Recognizers are not needed

As Section 2 shows, there is a need, e.g., adding a floating point recognizer or a dot-notation parser. Currently systems add these things through system-specific hooks; standardized recognizers would make it possible to do such things portably, and define and use them in portable libraries.

You may not see a need for all the features mentioned in Section 2, but if there is just one you need and that your vendor does not provide, the recognizers have paid off for you and for the vendor (who does not have to develop and maintain the feature himself).

3.2 People could misuse recognizers

People can already misuse a lot of things in Forth (e.g., `: 0 1 ;`), but Forth is not a nanny language. Forth design centers around responsible programmers, so while we will see some cases that most will consider misuses, it is much more important whether we will see some good uses. While not everyone will see all the uses mentioned above as good uses, as long as there are some that are considered good uses, it's a good reason to standardize recognizers. After a period of experimentation, there will be a rough consensus on what are good uses of recognizers and what aren't.

3.3 Recognizers are an attempt to make Forth more like C

C does not have a way to extend literals in a user-defined way, so, in a way, recognizers make Forth less like C. One could use recognizers to recognize some C lexical or small syntactic elements, and one can see the dot-notation recognizer as going in that direction. But note that people have been doing that even without standardized recognizers (in a non-standard way); also note that various people, including Chuck Moore, Julian Noble, and Andrew Haley have implemented infix notation or infix programming languages in Forth (something that recognizers do not facilitate), so if a Forth programmer is determined to go there, leaving recognizers away won't stop him.

3.4 Use parsing words! It's more Forth-like

Technically, parsing words cause problems: Either when trying to cut-and-paste between interpretation and compilation, such as `'/[']`, or in corner cases, such as `s"`. Recognizers avoid these problems and are therefore preferable. Indeed, one of the big advantages of recognizers is that they provide a long-term perspective for eliminating these problems.

As for *Forth-like*, recognizers for integers (singles and doubles) were part of Forth from the start. And simple ways to allow inputting dates and telephone numbers were part of the number recognizers of Forth, Inc. The only thing that was missing was the possibility to add user-defined recognizers; the use of parsing words, such as `s"`, is a workaround for this shortcoming, not a virtue.

4 Implementing a recognizer

This section looks at different implementation approaches for defining a recognizer. The outside interface of these implementation approaches can be made compatible, so these implementation techniques can both be used in the same system if desired.

4.1 Temporary words

A recognized string should behave like a word, so one way to implement a recognizer is to actually let it define a word when recognizing a string; e.g., for a string `123`, there is a temporary definition (not in any wordlist, name not important):

```
123 constant #123
```

and the xt of this word is `execute`d in interpretation state, or `compile,`d in compile state, like a regular word. It can also be `postpone`d or ticked.

However, one problem is that, in some of these uses, the word must be preserved and cannot be just temporary. After `execute`ing the word, we no longer need it and can reclaim the memory it uses; for `compile,`, it depends on how that is implemented. The classic threaded-code implementation (just `,`) would require that the word is preserved; however, many modern compilers have an intelligent `compile,` that compiles constants to literals, without reference to the compiled word, and therefore there is no need to preserve the word in that case. For `postpone` and ticking, the word must generally be preserved.

These words can be created in a separate section [Ert16], with the space reclaimed if the word does not need to be preserved.

There remains the problem of knowing whether the word needs to be preserved when the word is `compile,`d. The defining word could leave a flag in the defined word (or maybe a global flag) that indicates whether the word leaves a reference to itself when it is `compile,`d.

As a simple example, the core of a recognizer for unsigned single numbers looks as follows:

```
: (single-rec) ( c-addr u -- nt )
  0. 2swap >number 0= if \ it is a number
    2drop noname constant lastxt exit then
  2drop drop r:fail ;
```

`>Number` is used to check and convert the string, and if successful (0 unconverted characters), an unnamed constant is created and it's name token (nt) is returned; if unsuccessful, it returns a failure indicator (`r:fail`).

There is also a need to switch sections and perform other management tasks; these are always the same, so they are factored into a word `rec2-wrapper ( c-addr u xt -- nt )`, and the full recognizer is:

```
: single-recognizer ( c-addr u -- nt|0 )
    ['] (single-rec) rec2-wrapper ;
```

and this recognizer is added to the recognizer stack in the second position with:

```
get-recognizers
' single-recognizer -rot 1+ set-recognizers
```

4.2 RfD approach

The temporary word approach is relatively easy to understand and write, but it puts quite a number of demands on the system: The system needs to support another section[2], it must be able to create words in the middle of another word, possibly nameless or with their name coming from the stack, and ideally it should inline the code for that word when `compile,`ing it.

While most modern systems have these features or can add them without too much trouble, for standardization we may prefer an approach that puts fewer demands on systems and that does not require standardizing all the features that the temporary-word approach requires. The Recognizer RfD [Tru15] proposes such an approach.

Let's look at our example of a recognizer for unsigned single numbers again.

```
: comp-lit postpone literal ;

' noop     \ interpretation
' comp-lit \ compilation
' comp-lit \ part of postponing
recognizer: r:single

: single-rec ( c-addr u -- u2 rec | r:fail)
    0. 2swap >number 0= if \ it is a number
        2drop r:single exit then
    2drop drop r:fail ;
```

[2]If only interpretation and compilation of recognizers is supported, and compilation of the word created by the recognizer does not leave a reference to that word, then a buffer for one word instead of a full section is sufficient.

```
get-recognizers
' single-rec -rot 1+ set-recognizers
```

The recognizer `single-rec`[3] looks very similar to `(single-rec)` in the temporary word approach, but instead of putting the number in a newly-created word and returning that, the number is left on the stack and in addition `r:single` is pushed.

`R:single` is a word defined with `recognizer:` as a handle for the three actions, and the text interpreter (and `postpone`) access the actions they need through `r:single`:

- When interpreting, just leave u2 on the stack by `execute`ing `noop`.
- When compiling, compile u2 as a literal by `execute`ing `comp-lit`.
- When postponing, the final compile also happens with the compilation action, but one level later, so the compilation action `comp-lit` is `compile,`d. That requires that the data is transfered from the time when the recognizer runs to the time when the compilation action runs; to achieve that, the postponing part `comp-lit` is `execute`d before `compile,`ing the compilation action.

For literals, the usual pattern of the actions is `noop` for interpretation and the appropriate `literal` variant for both compilation and postponing.

The option for deviating from this pattern is useful for other applications of recognizers, such as replacing `to`.

The RfD does not specify the representation of `r:single`. In the current version of Gforth, this is implemented in a way that is compatible with the temporary-word approach: `r:single` returns the nt of a word, and you can get the interpretation action with `name>interpret`; you get the compilation action with `name>compile`; and there is also a field `>vtlit,` for the postpone-part action.

5 Recognizers where?

Recognizers can recognize words as well as numbers, so where should they hook in? There are at least three answers:

5.1 In `find`

(or its modern replacement, e.g., `find-name`). The benefits of this approach are:

[3]Matthias Trute would call it `rec:single`, but I find the presence of both "r:..." and "rec:..." confusing.

- It's a very natural fit for the temporary-word approach: `Find` returns a word, and so do temporary-word-creating recognizers.
- `Find` already has a way to add or remove things to be recognized: the search order. So recognizers could be added or removed from the search order, like wordlists, avoiding an additional mechanism. However existing code dealing with the search order may not be designed to deal with various recognizers on the search order.

The disadvantages are:

- Existing users of `find` (e.g., cross-compilers) would be surprised by `find` recognizing numbers, and the user's own number handling would be shadowed. That could be worked around by changing the search order appropriately when calling such users.
- For the RfD approach, the fit is not so great. In particular, we would now have a `find` that can generate additional values in addition to the xt (or nt for `find-name`) that it should produce. In many cases that is probably not a problem, but in some cases, it would be.
- Also, depending on the way words like `r:single` are implemented, and the actual `find` replacement that we want to hook in, there may be a mismatch; viewed differently, the implementation options for `recognizer:` would be restricted (but that is not necessarily a disadvantage).

5.2 In the text interpreter

The classical text interpreter first tries `find` and then tries numbers. In the current Gforth implementation, and in the text interpreter example given in the RfD, the `find` and number-handling parts are replaced by a recognizer-handling part. The search order search is performed by a word recognizer in the recognizer stack.

The advantages and disadvantages are the converse of those for the `find`-hooking approach:

Advantages: `find` users are unaffected, and the implementation of recognizers has fewer restrictions.

Disadvantages: We need the recognizer stack in addition to the search order (but don't need to worry about existing programs doing bad things to it).

5.3 As text interpreter hook

Instead of replacing the text interpreter the recognizer handling is added as a hook to the existing text interpreter. This would make the text interpreter more complex and reduce the options available to the programmer, so it offers only disadvantages, except that some consider it advantageous to reduce the options available to the programmer.

6 Other design decisions

There are some other design decisions where the right decision is not obvious, in particular: How to deal with recognizer stacks; whether to use `r:fail`, 0, or an exception as a failure indication. These and other design decisions are discussed at length in the RfD, which is recommended reading [Tru15].

7 History

System-specific hooks in the text interpreter have existed for a long time.

In 2003, Josh Fuller used *recognizer* in the sense used here, and proposed doing things like recognizing dates and (something like) dot notation by adding new recognizers[4]; the ensuing discussion points out that many systems have mechanisms for adding new recognizers. In that discussion, Jonah Thomas considered ways to deal with multiple recognizers, but not how to deal with interpretation, compilation, etc. in that context.

In 2007, in a discussion about number parsing hooks, I sketched some ideas about recognizers `news:<2007Aug4.093801@mips.complang.tuwien.ac.at>` `news:<2007Aug4.161609@mips.complang.tuwien.ac.at>`. I did not pursue this idea further at the time, but Matthias Trute picked it up and proposed using it for a dot-parser `<0dgjs6-h4e.ln1@wolf.stein.zeit>`, and implemented them in amForth [Tru11]. Subsequently, they were also implemented in Win32Forth, Bernd Paysan implemented them in Gforth [Pay12a, Pay12b], and Matthias Trute made a Forth 200x RfD [Tru15] proposing standardization.

8 Conclusion

User-defined recognizers generalize the Forth number recognizer and various system-specific hooks in the text interpreter. They allow to replace parsing words and their problems (e.g., `state`-smartness), write a dot-notation parser, and have other benefits. While a number of people have argued against user-defined recognizers, I have not seen a technical argument against them yet.

For implementing a recognizer, we look at two options: Creating a temporary word is a little easier

[4] `http://compgroups.net/comp.lang.forth/additional-recognizers/73467`

to understand, and you get correct `postpone` behaviour for free, but it requires more infrastructure from the system, in particular support for a section for these temporary words and knowledge about whether `compile,` produces a reference to the temporary word. The other option, defining interpretation, compilation, and postponing behaviour is a little harder to understand, but not much longer, and it requires less infrastructure from the system. The latter approach has been proposed for standardization and is preferable for this purpose.

Another design decision is whether to hook into `find` or into the text interpreter. While hooking into `find` has some advantages, the advantages of hooking into the text interpreter, in particular with respect to backwards compatibility, outweigh them.

References

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997. 2.6, 1

[Ert98] M. Anton Ertl. `State`-smartness — why it is evil and how to exorcise it. In *EuroForth'98 Conference Proceedings*, Schloß Dagstuhl, 1998. 2.4

[Ert16] M. Anton Ertl. Sections. In *32nd EuroForth Conference*, pages 55–56, 2016. 4.1

[Mah98] Manfred Mahlow. Prelude and finale: Implicit context switching based on pre- and post-executed words. In *14th EuroForth*, 1998. 2.6

[MPE16] MPE. *VFX Forth for x86/x86 64 Linux*, 4.72 edition, 2016. 2.6

[Pay12a] Bernd Paysan. Recognizer. *Vierte Dimension*, 28(2):37–38, 2012. 7

[Pay12b] Bernd Paysan. Recognizers. In *28th EuroForth Conference*, pages 108–110, 2012. 7

[Tru11] Matthias Trute. Recognizer — interpreter dynamisch verändern. *Vierte Dimension*, 27(2):14–16, 2011. 7

[Tru15] Matthias Trute. Forth recognizer — request for discussion. 3rd RfD, Forth200x, 2015. 4.2, 6, 7