Poor Man's Recognizer

µForth is the cross compiler for µCore. When I started the project on top of gforth_062, I found a simple and powerful solution to cross compilation by patching the way `[` and `]` behave. The current version of gforth_079 uses Matthias Trute's recognizer mechanism and I gave them a try as an alternative to patching.

The result of this endeavor was so complicated that I returned to the patch solution, which I regard as much more readable. It can be implemented on all versions of gforth above 062 albeit minute differences, because gforth's interpreter/compiler is a moving target. Alternatively, the patch version can be implemented in a way that it reads like recognizer code, though drastically simplified.

In order to clarify the issues, I will first present and discuss the patch solution, followed by the gforth recognizer solution, and finally followed by the "Poor Man's Recognizer" solution.

µForth has two very different modes of operation:

1. `host-compile` compiles into the host's dictionary in order to add capabilities to µForth.

2. `target-compile` compiles into the target's dictionary in order to produce code for µCore.

These two modes require very different parsers, which are both different from the native gforth parser, because µForth includes an OOP package for the sake of operator overloading (i.e. a `@` may do very different things depending on the object that preceded it). This would be straightforward, if `[` and `]` were deferred, but they are not and therefore, `[` and `]` have to be patched instead. For the sake of argument, I will only discuss the `target-compile` mode.

1 Patching

The patch magic is done in a portable way by `becomes`, which takes an xt and the name of an existing word:

```
: becomes ( <word> new-xt -- ) \ make existing <word> behave as new-xt
   >r   here   ' >body dp !
   postpone AHEAD   r> >body dp !   postpone THEN
   dp !
;
Variable 'interpreter   ' interpreter 'interpreter !
Variable 'compiler      ' compiler    'compiler !

:noname ( -- )  'interpreter @ IS parser   state off ;   becomes [
:noname ( -- )  'compiler @    IS parser   state on  ;   becomes ]
```

After patching, everything still works ok, because we initialized both `'interpreter` and `'compiler` with gforth's native code. Now we can define the new parser for the target.

First some lower level words:

`d>target ( d.host -- d.target )` converts a host's double number into a double number for µCore, because very often the data width of the host is greater than µCore's data width.

`ClassContext`, a variable, points to a linked list of methods. If it is zero, Forth's dictionary will be searched. `search-classes` searches an object's list of methods down its inheritance order.

`debugger-wordlist`, a variable pointing to a wordlist of commands, which will be searched with preference when interactively debugging the target system.

`t_lit, ( n -- )` compiles a number on the host's stack as a literal in the target system.

`>t` transfers a number on the host's stack to the target's stack.

`not-found ( addr len -- )` displays the string that didn't match and aborts.

Now we are prepared to define the target's interpreter/compiler classical style, everything in one single definition:

```
: target-compiler ( addr len -- )
\ search for methods
   ClassContext @ IF  2dup search-classes ?dup
                      IF  nip nip name>int execute  EXIT THEN
                      not-found
                  THEN
\ search for commands while debugging
   dbg? IF  2dup debugger-wordlist search-wordlist
            IF  nip nip name>int execute  EXIT THEN
        THEN
\ search the target's dictionary
   2dup find-name ?dup IF  nip nip name>int execute  EXIT THEN
\ try to convert to an integer number
   2dup 2>r snumber? ?dup 0= IF  2r> not-found  THEN  2rdrop
   comp? IF  0> IF  d>target swap t_lit,  THEN  t_lit,  EXIT THEN
   dbg?  IF  0> IF  d>target swap >t      THEN  >t      EXIT THEN
   drop
;
```

When an object has been compiled or executed, `ClassContext` will have been set to its methods list to be searched. If a method is found, it will be executed producing target code and we are done. If nothing was found, we display the not-found message and abort.

Otherwise, we check whether we are interactively debugging, in which case we will search `debugger-wordlist` next. If it was a command, it will be executed and we are done. If it was no command, we will search the target's dictionary. If it was a target word, it will be executed producing target code and we are done.

If it was neither a command, nor a target word, `snumber?` will try to convert the string into a number. When successful, we may be

1. compiling. In this case, we have to compile the number as literal(s) into the target code and we are done.
2. debugging. In this case we transfer the number on the host's stack to the target's stack and we are done.
3. interpreting. In this case we throw away the double number flag and we are done, leaving the number on the host's stack.

You must admit, this is VERY different from what the native gforth compiler does.

Finally, we can define

```
: target-compile ( -- )
   ['] target-compiler dup 'interpreter !   dup 'compiler !   IS parser
   Targeting on   Only Target also
;
```

that will activate the `target-compiler`, set `Targeting` to true and set the dictionary search order for target words.

2 Recognizers

Ok, now gforth's recognizers. The lower level words we need have already been explained above.

The debugger words are handled separately, so we define our first recognizer:

```
: rec-debugger ( addr u -- nt rectype | rectype-null )
   dbg? IF  debugger-wordlist find-name-in
            dup IF  rectype-name  EXIT THEN  dup
         THEN  2drop   rectype-null
;
```

Hm, we see that we have to do more than just define a recognizer, we also have to define recognizer types like `rectype-name` and `rectype-null`, which have been defined in gforth_079.

Now come the methods and the Forth dictionary. Remember, no matter whether we are in interpret or compile mode, we always execute the command of the target code compiler. So we have to define our first rectype, because the standard one, `rectype-name`, does not do what we need.

```
:noname   name>int execute-;s ;  \ interpret action
dup                              \ compile action, same as above
' lit,                           \ postpone action
rectype: rectype-target
```

Now we can define the recognizer for methods compilation,

```
: rec-methods ( addr u -- nt rectype | rectype-null )
   ClassContext @ 0= IF  2drop rectype-null  EXIT THEN
   2dup search-classes ?dup IF  nip nip rectype-target  EXIT THEN
   not-found
;
```

and the one for the Target's dictionary:

```
: rec-target ( addr u -- nt rectype | rectype-null )
   find-name ?dup IF  rectype-target  EXIT THEN  rectype-null
;
```

Ok that takes care of the methods and normal Forth words. Now we must turn to numbers. Again, we can not use the normal `rectype-num` and `rectype-dnum`, because we are compiling code for the target.

```
' noop
:noname ( n -- )  comp? IF  t_lit,  EXIT THEN  dbg? IF  >t  EXIT THEN  drop ;
dup
rectype: rectype-tnum

' noop
:noname ( d -- )  d>target swap
   comp? IF  t_lit, t_lit,  EXIT THEN
   dbg?  IF  >t     >t      EXIT THEN  2drop
;
dup
rectype: rectype-tdnum

: rec-tnum ( addr u -- n/d table | rectype-null )
   snumber? ?dup 0= IF  rectype-null  EXIT THEN
   0> IF  rectype-tdnum  ELSE  rectype-tnum  THEN
;
```

Now we have a single/double integer recognizer for the target and we are prepared to define the recognizer for target compilation consisting of four single recognizers:

```
$Variable target-recognizer
align here target-recognizer !
4 cells , ' rec-tnum A, ' rec-target A, ' rec-debugger A, ' rec-methods A,
```

Finally we can define:

```
: target-compile ( -- )
   target-recognizer TO forth-recognizer
   Targeting on   Only Target also
;
```

that will activate the `target-recognizer`, set `Targeting` to true and set the dictionary search order for target words.

Frankly, I find the patch version of the first chapter a lot easier to read.

Why is this so?

In the patched version, everything is close together in one definition: From string through wordlist lookup to the final execute. Therefore, you may modify everything when needed. We could even do without patching, if `[` and `]` were deferred. Its drawback: Everything is merged into a single definition with a more or less complex conditional structure.

The recognizer version gets rid of the complex conditional structure, but it has newly invented complexities: Separation of the search action from semantic interpretation (rectypes), and the final execute is completely hidden. Therefore, in order to understand the recognizer code, you must make yourself a mental image that merges the search action and its semantic interpretation, and you better understand the underlying recognizer machine that does the final execute for you. Let alone the definition of the final -recognizer, which is not a colon definition but a data structure, which has to be read backwards. Challenging conditions for reliable code that ought to be easy to understand.

3 Poor Man's Recognizer

As a compromise, here is a simplified version for a recognizer type construct, which does not need all the overhead of gforth's recognizer mechanism.

Foremost, we want to get rid of the convoluted control structure of the patch version. I.e. we want to be able to write down an easy to read specification for `target-compiler`, which behaves identical to the patched version of the first chapter as follows:

```
: target-compiler ( addr len -- )
   method-find debugger-find target-find target-number not-found
;
```

In the Recognizer chapter we finally defined `target-recognizer` as a reverse list of simple sub-recognizers, each of which only does a single string matching attempt.

Instead, the simplified version defines a colon definition, which basically does the same thing. It has the limitation that it can not change its behaviour dynamically. But who needs this flexibility? After all, the parser has a much lower change rate compared to the dictionary's search order.

Each sub-recognizer has identical stack behaviour:

It takes a counted string as input argument.

If there is no match, it leaves the counted string unchanged handing it over to the next sub-recognizer.

If there is a match, the input arguments are dropped, the return address into `target-compiler` is thrown away and therefore, we continue to execute the word that called `target-compiler`.

If none of our sub-recognizers did match, we bump into `not-found`.

These are the definitions of the sub-recognizers:

```
: method-find ( addr len -- addr len | rdrop )
   ClassContext @ 0= ?EXIT
   2dup search-classes ?dup
   IF  rdrop nip nip   name>int execute  EXIT THEN
   not-found
;
: debugger-find ( addr len -- addr len | rdrop )
   dbg? 0= ?EXIT
   2dup debugger-wordlist search-wordlist
   IF  rdrop nip nip   name>int execute  EXIT THEN
;
: target-find ( addr len -- addr len | rdrop )
   2dup find-name ?dup IF  rdrop nip nip   name>int execute  THEN
;
: target-number ( addr len -- addr len | rdrop )
   2dup 2>r   snumber? ?dup 0= IF  2r> EXIT THEN  2rdrop rdrop
   comp? IF  0> IF  d>target swap t_lit,  THEN  t_lit,  EXIT THEN
   dbg?  IF  0> IF  d>target swap >t      THEN  >t      EXIT THEN
   drop
;
```

For completeness, here is the host's interpreter/compiler:

```
: host-find ( addr len -- addr len | rdrop )
   2dup find-name ?dup 0= ?EXIT   rdrop nip nip
   comp? IF  name>comp  ELSE  name>int  THEN   execute
;
: host-number ( addr len -- addr len | rdrop )
   2dup 2>r snumber? ?dup 0= IF  2r> EXIT THEN   2rdrop rdrop
   comp? IF  0> IF  swap postpone Literal  THEN  postpone Literal  EXIT THEN
   drop
;
: host-compiler ( addr len -- )   host-find host-number not-found ;
```

4 Summary

I am sorry to say, but Matthias Trute's recognizer mechanism is over-engineered and therefore, it is difficult to understand, explain, and extend. An indication for this is the sheer number of articles, which try to explain it. In addition, it is more flexibel than actually needed.

The "Poor Man's Recognizer" had been a vague idea for several years. After writing the first two chapters of this paper, I finally implemented it. It exceeded my expectations as far as complexity is concerned and therefore, this is what I will add to my Forth toolbox.