Poor Man's Compilers

how Forth treats its source code

The way Forth processes its source code is peculiar and unfortunately quite unknown. The compilation in particular is unusual. It is extremely simple and allows the syntax to be expanded on the fly. For specific application situations, specific syntactic extensions can be defined as "syntactic sugar", which make the source code easier to read and therefore more reliable.

1 How Forth came about and where the name comes from

Forth was developed by Charles "Chuck" Moore, who had worked as a freelance programmer since the late 1950's. Primarily in automation projects with real-time requirements. First on mainframes, since the mid-1960s on minicomputers, which at that time often only offered an assembler as software support.

Over a 10-year period, he experimented with different programming concepts to increase his productivity as a programmer. Finally, in 1968, he realized that he had developed a complete programming language, which he wanted to call "Fourth" by the time the third generation of computers was talked about. But the assembler he used only allowed five-letter identifiers. So it became "Forth".

Chuck Moore: "I developed Forth over a period of some years as an interface between me and the computers I programmed. The traditional languages were not providing the power, ease, or flexibility that I wanted."

2 The Forth System

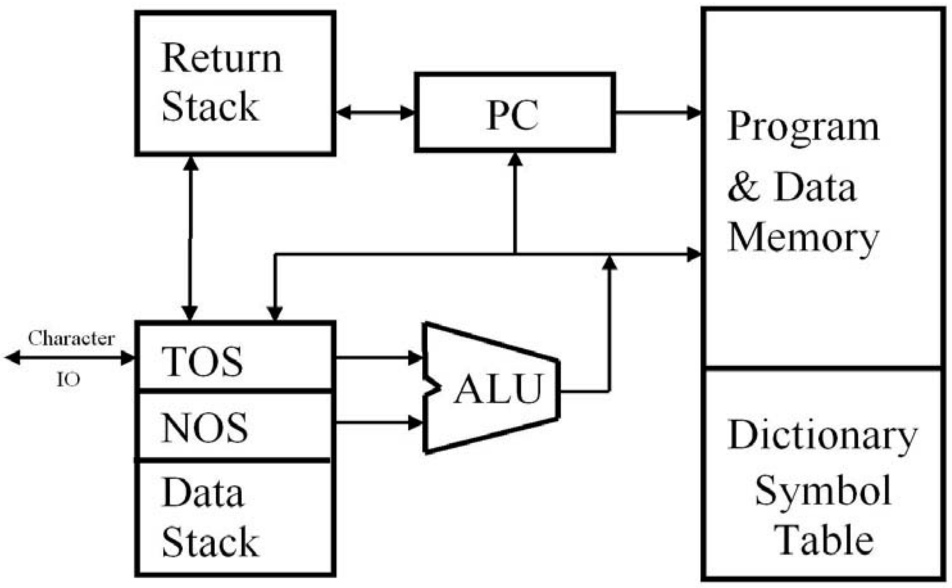

Fig 1: The Forth Maschine

The language is based on a virtual machine[1] as a programming model, with Forth as the "Assembler" of this machine.

The Forth machine has no registers, instead it has two stacks. The data stack serves the calculation of results. The return stack for storing return addresses of function calls and within a function, which is called "word" in Forth, additionally as loop counter and temporary memory.

When a function is called, the input parameters are on the data stack. At the end of the call, these are replaced by the function results. The ALU inputs are connected with TopOfStack (TOS) and NextOfStack (NOS), the ALU output leads back to TOS. Words for stack manipulation allow to control the sequence of parameters on the data stack in order to modify (**`swap`**), or duplicate (**`dup`**), or destroy (**`drop`**) parameters. For arithmetic expressions, Forth uses reverse Polish notation[2], which needs getting used to. Because of the stacks, Forth is a null-address machine, and functional style programs can be written.

Two stacks are required because the dynamics of the two stacks are different. A word such as **`+`** consumes two input values and leaves one result on the data stack. When a function is called, the return address is placed on the return stack and must be there again as top element for the **`EXIT`** at the end of the function[3].

The PC is used to address the sequential program code in memory.

The dictionary is a tree structure of word lists that contain the commands of the Forth machine. Humans and other systems communicate with the Forth machine via ASCII encoded Character IO.

[1] Due to its simplicity, it can be efficiently implemented as a "real machine".
See https://github.com/microCore-VHDL

[2] Older engineers still fondly remember HP's calculators.

[3] The runtime environments of most programming languages have only one stack. This complicates the handling of return addresses and data and led to the invention of "stack frames", which, however, significantly increase the cost of a function call at runtime..

3 The Dictionary

A central element of Forth is the symbol table, which is called dictionary and is still available after compiled as a principle. That's why Forth code can be executed interactively over a Character-IO interface.

The dictionary is a tree structure of word lists, the so-called vocabularies. The root of the dictionary is the `Forth` vocabulary that contains the basic words of a Forth system[4]. After the lexical analysis, the `Context` is searched, a dynamically configurable list of vocabularies.

New word definitions are entered at the top of the `Current` word list. That's why it is possible to redefine already existing entries under the same name, e.g. to add debugging features.

Because of the vocabulary structure, identical word names can appear in different vocabularies that have different semantics. Which semantics apply at any point in time is well defined by `Context`.

In addition to the word name, each word list entry contains a reference to executable code that defines the semantics of the word.

4 Parsing in Forth

The lexical analysis is very simple: each character string surrounded by spaces (blank, tab, cr etc.) is a token. That's basically how we read text[5]. Not only is this easy, but it also has a collateral benefit: names in Forth may contain any ASCII character except spaces[6].

In the next step, the parser searches the word lists of `Context` to see whether the token exists in the dictionary. If so, its associated code is executed. If not, it is checked whether the token is a number. If so, the number is pushed on the data stack.

If it is neither found in the dictionary, nor a number, then the error message `?` displayed.

This is how the Forth system works in interpretation state.

To debug a word, numbers are placed on the stack as input parameters (if the word has any input parameters at all), the word to be examined is executed by entering its name and then the contents of the stack are displayed, which should now contain the output parameters of the word at the top of the data stack.

[4] Forth is standardized as ISO/IEC 15145:1997. The standard is continuously developed, see: https://forth-standard.org.

[5] Why do most programming languages do it differently? Two reasons have occurred to me so far:
1. Fortran was geared toward mathematical formulas, and mathematicians traditionally omit the spaces around operators.
2. Columns on the punch cards were "expensive" and therefore the strategy serves a primitive data compression.
In any case, the result is that the "Fortran" method was copied without questioning and gave us RegEx. And the programmers have to live with annoying restrictions on naming.

[6] Therefore, special characters can be used in names to provide semantic hints. E.g. `"` for string functions, `.` for output on the display, `#` in the first position for constants.

5 Compilation in Forth

The system variable `State` determines whether the Forth system is in interpretation state (`State = 0`) or in compilation state (`State = 1`).

5.1 Colon und other Compilers

The following line is input to the Forth system:

```
: xlerb word1 word2 word3 ;
```

`:` creates a new entry in the `current` vocabulary named `xlerb` and sets `State` to 1, i.e. the system switches to compilation state. This is followed by the words `word1`, `word2` and, `word3`, which must already exist in the dictionary. They are now not executed but compiled as function calls. This sequence of executable code is associated with `xlerb` and represents its semantics. The `;` has the immediate bit set (see 5.2) and is therefore executed instead of being compiled as function call. It compiles an `EXIT` and resets `State` to 0. This completes the definition of `xlerb` and the system is back in interpretation state.

Besides `:` there are other words that create vocabulary entries but do not enter compilation state:

`Variable <varname>` reserves a memory cell and puts its absolute address on the data stack when `<varname>` is subsequently executed.

`<number> Constant <constname>` pushes `<number>` onto the data stack if `<constname>` is subsequently executed.

`Vocabulary <vocname>` creates the new vocabulary `<vocname>`.

5.2 The Immediate Bit

The immediate bit is a marker bit in the name of a word entry in the dictionary[7]. If set, then that word will also be executed in compile state instead of being compiled as a function call. So an immediate word is a small compiler. The `;` we have already met above.

The immediate bit now makes it possible to create control structures. An example:

```
: conditional ( flag -- ) IF word4 ENDIF word5 ;
```

`:` creates the word entry `conditional`.

The string in parentheses is a comment[8] and means that `conditional` has one input parameter `flag` and no output result.

This is followed by `IF`, whose immediate bit is set and is therefore executed. The `IF` compiles `0=branch` and reserves space for the branch target address, which is not known at this point. Therefore, a 0 is temporarily entered there during compilation. And its address is placed on the data stack. The compiler `IF` is already finished.

`Word4` follows, which is compiled as a function call.

Then comes `ENDIF`, another compiler. The previously missing target address of the preceeding `IF` is now known and is entered after `0=branch` at the address that is on the stack, because `IF` put it there. This completes `ENDIF`.

[7] The immediate bit is set in that the `;` of a colon definition is immediately followed by the word `immediate`.

[8] Because of reverse Polish notation, the brackets are not used for arithmetic expressions, so they can be used for comments.

The function call of **`word5`** follows and then **`;`** finishes the definition of **`conditional`**.

If there is a 0 on the stack when **`conditional`** is executed, then **`0=branch`** consumes this flag and jumps directly to **`word5`**.

If there is a number /= 0 on the stack, then **`0=branch`** also consumes this flag, but no branching occurs, and **`word4`** and **`word5`** are both executed.

In Forth there is hardly any predefined grammar and no compiler that checks the source code for compliance with grammar rules. Instead, there are individual small compilers that interact synergistically and thereby generate syntax. Additional compilers can be defined as part of an application. Forth is not only expandable in terms of its range of functions, but also in terms of its syntax.

Below is a striking example.

6 Syntactic Sugar

ASCII text is to be output as Morse tones. To do this, a Morse code table must be created. And that's potentially error-prone.

Here is a "desired syntax" that came about after a few mental iterations between "easily realizable" and "well readable". This is followed by the code for the compilers[9] required for this.

```
morsetable:
   . _     | A      _ .      | N
   _ . . . | B      _ _ _    | O
   _ . _ . | C      . _ _ .  | P
   _ . .   | D      _ _ . _  | Q
   .       | E      . _ .    | R
   . . _ . | F      . . .    | S

   and so forth ...
;morsetable
```

[9] In the executable system it has to be exactly the other way round, because Forth traditionally compiles its source code in one pass and therefore words that are used must have been defined beforehand.

And this is what the compilers for the desired syntax look like:

```
$80 cells Constant #codes \ size of the code table

Create Morsetable  #codes allot   Morsetable #codes erase

Vocabulary <morse>

: morsetable: ( -- 0 )
   also <morse>           \ make <morse> first vocab. in CONTEXT
   0                      \ end marker for the first code
;

also <morse> definitions   \ set CURRENT to <morse>

1 Constant .
2 Constant _

: morsecode!  ( count bits <character> -- )
   swap 8 lshift or       \ pack count and code into 16 bit value
   char cells
   dup #codes < 0= abort" character out of range"
   Morsetable + !         \ store at char's position
;
: | ( 0 n1 .. nr -- 0 )
   0 ( count )  0 ( morsebits )
   BEGIN  rot ?dup
   WHILE
      1- swap 2* or       \ add next morsebit
      swap 1+ swap        \ and increment count
   REPEAT   morsecode!
   0                      \ end marker for the next code
;
: ;morsetable ( 0 -- )
   0 - abort" malformed morse table"
   previous               \ remove <morse> from CONTEXT
;
previous definitions      \ reset CURRENT to previous vocabulary
```

The vocabulary **`<morse>`** now contains the words **`.`**, **`_`**, **`|`**, **`morsecode!`** and **`;morsetable`**.

7 Bibliography

C.H.Moore: "Forth: A new way to program a mini-computer", 1974,
Astron. Astrophys. Suppl. **15**, 497-511.

Leo Brodie: "Starting Forth", 1981,
https://www.forth.com/wp-content/uploads/2018/01/Starting-FORTH.pdf

Leo Brodie: "Thinking Forth", 1984,
https://www.forth.com/wp-content/uploads/2018/11/thinking-forth-color.pdf