CHAPTER 1. INTRODUCTION TO LaFORTH

1.1. INTRODUCTION

LaForth was developed by LaFarr Stuart and Robert L. Smith, concurrently with figForth in 1978-79. Both of them were actively participating the Forth Implementation Team which released the figForth Model on 6 different microprocessors in 1979. However, LaFarr was not satisfied with the figForth Model. He put his many ideas into LaForth and used it to demonstrate the results of his experimentations. In one of the FIG meetings, he jokingly introduced himself by announcing that: "I mutilating Forth".

Here we will let the principal author of LaForth, LaFarr Stuart tell the story of LaForth. The text was published in the Proceedings, 1980 FORM Conference at Asilomar, California, November 26-28, 1980.

"First and foremost LaForth is a toy. It is not well defined; I change it whenever I get a bright idea. It does not have any other users so there is no obligation to be consistent with past versions, or previous documentation. It is a variant of Forth, but with no obligation to follow or conform to Forth. I made changes and put in features simply because I wanted them; I left out features often out of laziness and/or stubbornness. I apologize but once in a while I am guilty of putting in a feature for "showoff" value.

I should point out that without Forth there would never have been a Forth Interest Group, and without FIG I would have not known about Forth. The people who created FIG deserve credit for giving all of us a very well thought out product and starting point. Riding the the FIG meetings with Bob Smith and Dave Lion provided a great setting to bandy ideas about. Then Bob Smith and myself would spend hours on the phone between FIG meetings.

No doubt the easiest change to make to any language is to simply relax some of the restriction. Much of what follows is simply that, but hopefully you will like some of the changes.

When I first saw Forth, there were a lot of things I thought I could improve. The most blatant was three character names. Many years ago when working on a FORTRAN compiler, I decided that I would never implement a language which limited names to fewer than 80 characters. (That decision was made in the days of card input.) Other things I felt should go the way of the Dodo bird were:

1. *Limited input line length.*
2. *Limit of 64 byte branches.*
3. *Limit of less than 64 characters in input strings.*
4. *Screens wasting 1024 bytes on disk independent of the amount of information they contain.*
5. *Screens on an output terminal, 16 by 64 characters only.*
6. *OK cluttering up the page.*
7. *Carriage return key not doing a carriage return. This deserves much more discussion. It involves the psychology of system design. The essence here is that the user must feel he is driving the system and he is in control; otherwise, he will feel driven by the system and be insecure.*
8. *No input should be lost. This implies interrupt driven I/O.*
9. *IF, THEN, ELSE, END, FLUSH, and probably other words violate conventional usage.*
10. *Crashes if you do a carriage-return after FORGET, :, CONSTANT, VARIABLE, VOCABULARY, and probably others. Crashes if you use the R-stack outside a : definition.*
11. *More than one address the user must be aware of associated with each word.*
12. *Typing errors wasting memory with "SMUDGEd" garbage.*
13. *No "soft" warning when the stack is getting full.*
14. *No direct way to write a program which requests a numeric value.*
15. *No facility to allow the user to stop a running program, and then resume execution of that program. APL calls this a "pending function".*

I should give you more detail on how I provide the ability to call LaForth from within a colon definition. This allows the user to use all of LaForth and even create new words then return to the routine which called LaForth.

This also allows a simple way to get a numeric value from the user; and allows keyboard action to suspend the execution of a function.

LaForth differs significantly from the figForth model in that execution occurs on the occurrence of a space or carriage-return; to accomplish this, INTERPRET is very different and uses different primitives. Some of the primitives of INTERPRET use the same names as the figForth model, but these words do things differently. Consequently, I will tell you logically how it is done, but you will have to adapt the idea to work in your environment.

In LaForth there is only one name which has a nonprinting character. It is the "backspace" character used all by itself. Remember, LaForth executes on the fly and the "Backspace" character is a perfectly acceptable word name. "Backspace" is the word which returns control from FORTH, to the word which called FORTH.

Here is how "Backspace" works. Somewhere within the execution of INTERPRET the word EXECUTE gets executed. At the time when EXECUTE executes "Backspace", "Backspace" removes from the R-stack the address which returns control to the infinite loop of FORTH, thus control returns to the word which called FORTH, and the deed is done.

To allow for key-board suspension of a program, I have a word KIN (keyboard interrupt). KIN test to see if there are any characters in the circular input buffer and if there are it executes FORTH. That is all there is to allow "pending functions".

One other significant difference in LaForth is in the names used for the conditional and looping words. Except for I/O, the control structures of any language is generally the most difficult to understand. It accounts for most bugs. It is most troublesome when trying to follow and understand the logic of a program."

Then LaFarr went on talking about the graphical control structure words in LaForth. The discussion will be discussed more fully in Section 1.3.3.

Originally LaForth was implemented on a 6809 microprocessor. Recently, it was moved to an 8088 under MS-DOS. As the Silicon Valley FIG Chapter is looking at various approaches to produce a Forth Model which can be readily adapted to the more powerful microprocessors of the 1990's, it becomes apparent that many of the features in LaForth are very useful. It is thus appropriate to publish LaForth in a form useful to people who might want to help producing this new Forth Model and actually move it to different processors.

1.2. A GUIDED TOUR OF LaFORTH

LaForth is distributed on a 5.25" MS-DOS 360K diskette. It contains the following files:

READ.LA	Boot instructions
LA.ASM	Assembly source listing
LA.COM	LaForth execution file
LA.HI	High level utilities
LA.SUM	Summary of LaForth glossary
LA.DEF	LaForth glossary

Assuming that you are using this diskette in Drive A, type

```
A>LA
```

will bring up the LaForth kernel. The kernel is not very useful because it has a very limited vocabulary, just enough tp support the text interpreter. To load LaForth with all the utilities, type

```
A:>LA LA.HI
```

This DOS command boots LaForth and also opens the LA.HI files, which contains all the high level commands to build a fully functioning LaForth system. After LaForth is loaded, type

```
LT RUN
```

You will see a few messages indicating re-definition of some words, and a simple OK message. That will be your last prompt! Here on words typed will be executed immediately after a space or a carriage-return is pressed. If the word is executed satisfactorily, the cursor will move to accept the next word. If a word is not found in the dictionary, LaForth will beep and the cursor is backspaced to the beginning of the last word. You can type the correct word over to get it executed.

To see a list of words in La-Forth, type

```
.DICT
```

followed by a space (or carriage return). To stop the display, just hit any character.

Another word DICT is like .DICT, but prints the names with their addresses.

DUMP is a word which displays the contents of a block of memory. It takes two arguments on the stack: the starting and the ending addresses of the memory block. An example is:

```
HEX 100 200 DUMP
```

which displays the first 256 bytes of LaForth object code.

An interesting command sequence is:

```
LT .TEXT
```

LT puts on the stack the segment and the starting address of the file text buffer, and .TEXT prints out the entire file. Since LA.HI file was just opened and read into this file, that's what you see.

A fun word is GCD, the greatest common divisor. Type two numbers followed by GCD and two periods, like:

```
12345 7890 GCD . .
```

You will see two numbers 0 and 15 displayed. 0 is the remainder and the common divisor is 15.

Every want to know which day in the week is your birthday? Enter your birthday in the form DD MM YY and the following commands:

```
7 4 51 CDN 7 MOD .
```

CDN converts the date April 7, 1951 to the Julian date: 18787 days after January 1, 1900. It's modulus of 7 is 6, which means Saturday.

For those interested in how fast LaForth runs, type

```
PRIME
```

and LaForth will report "720 PRIMES" in about a second. The time depends on your machine and its clock rate. PRIME does the standard Sieve of Erostotheses test.

When you are tired of LaForth, type

```
BYE
```

and you will be back to DOS.

La-Forth is a very interactive system. Any word will be executed as soon as it is terminated by either a space or a carriage return.

A detailed list of commands in La-Forth can be obtained from the file LA.DEF . Summary information can be found in LA.SUM. For real details, see the assembly language source LA.ASM and the high level definitions in LA.HI .

1.3. SPECIAL FEATURES IN LaFORTH

1.3.1. IMMEDIATE EXECUTION

Most Forth systems process keyboard input on a line-by-line basis; i.e., they read in a whole line of text and only process the text after a carriage return is entered to signify the end of a line. This is a hangover from the very old days when the computer knew only how to deal with punched cards, which holds up o 80 characters of text. With personal computers, which are designed to give their full attention to their owners, forcing the computer to sleep while the user is typing in a line of text is a pure waste of its power. The computer can and should provide much more service to the user, and help him with his typing as much as it can.

LaForth intensely looks at the characters the user types in. Whenever a word delimiting character, a space or a carriage return, is detected, it immediately goes to work. If the word can be found in the dictionary, it is executed or compiled immediately. If the word does not exist in the dictionary and it can be converted to a number, it is converted and the resulting number is pushed on the data stack or compiled as a literal into the dictionary. If the number conversion failed, LaForth cannot do anything about the word, and it beeps and moves the cursor back to the beginning of the offending word to let the user typing in the correct word.

LaForth thus processes the text one word at a time. The spaces have the same effect as the carriage returns. They all cause LaForth to process the word just finished. The only difference is that a carriage return causes an additional line-feed character echoed back to the terminal.

Not waiting for a whole line of text allows LaForth to detect typing errors early, and thus helps the user to use the computer more efficiently. The interaction between the user and its computer is much more intimate and intense. Most often the user gains more confidence on the computer and the productivity is improved.

1.3.2. TEXT FILE AS MASS STORAGE

Since LaForth treats carriage returns similarly to spaces, it can process regular line-based text files without much additional efforts. Basically, the carriage returns in a text file can be considered as equivalent to spaces. Since LaForth stops processing a text string which it reaches a NULL character, text files can be terminated by NUL characters, as well as EOF characters.

The same text interpreter is used to process text entered from a keyboard and the text obtained from a text file, the design of the text interpreter is simple and fast. Most text files generated by popular word processors and editors can be used by LaForth for execution and compilation. Therefore, it is not necessary to burden LaForth to provide a text editor, which is an utility conventional Forth system must provide to deal with text stored in the block format. Since the block format is not commonly used by most commercial editors and word processors. Forth systems must provide the functionality, which is not a trivial application. Most Forth block editors lack the sophistication and user friendliness which are common expectation from the user's point of view. A block editor with limited functionality is the most common source of complains and irritation to the unsuspecting users.

1.3.3. LOOPS AND CONDITIONALS IN LAFORTH

For many years, LaForth has been a private and highly experimental version of Forth, used and modified only by us. There have been only a very few published papers and talks about LaForth, in part because we did not wish to conflict with figForth or the standardization effort. Some of the results of LaForth have, nevertheless, had some influence on other Forth systems. We have recently agreed to release a version of LaForth for others to use. We anticipate that its main use will be for experimentation. The strength of Forth is its simplicity, and in that sense we believe that LaForth is more Forth-like than Forth! It is very easy to modify. It is an excellent test bed for new ideas. It is a "lean and mean" type of a Forth system, in contrast to some of the recent "fat" Forths which have come into vogue.

One of the obvious differences of LaForth from other versions of Forth is a new set of words for conditionals and loops. The usual Forth words for conditionals like IF , ELSE and THEN are, at the least, quite confusing for beginners in Forth, whether or not they have learned another computer language previously. Words like BEGIN and UNTIL show nesting so poorly that many Forth programmers use indentation and lots of "white space" to clarify the nesting. In LaForth, the nesting tends to be quite obvious through the use of the left and right brackets, "[" and "]". The bracket symbols are quite well known typographical symbols clearly suggesting closure. The set of words in this paper appear to us to be far more consistent and readable than those currently in vogue. If you disagree, you can obviously replace our suggested words with any other of your choice.

The word [[can be used to mark the beginning of structures previously started by the word BEGIN or Eaker's CASE. The word]] is our suggested replacement for the words AGAIN and REPEAT . When used with no related conditionals it causes an indefinite repeat back to the [[.

Three of the LaForth words have the "?" symbol immediately preceding a bracket symbol. This indicates a test of the value on the top of the stack. Two other words use the ? symbol after a bracket to indicate an obvious "matching." We have tried to be consistent with the use of the "?" when used in conjunction with the bracket symbols. When used for matching at the end of a conditional, it merely resolves the implied forward references and does not imply any branching back.

Thus the usual IF ... ELSE ... THEN structure turns into the sequence

```
?[ ... ][ ... ]?
```

where the ... indicates the true and false parts. The symbol ?[can be used not only to replace IF but also the word WHILE in either its single or multiple forms. The "multiple while loop" has the form:

```
[[ ... ?[ ... ?[ ... ?[ ... ]]
```

where the final word]] causes a branch back to the [[mark and also resolves the forward conditional branches. An alternative form (the "andif conditional") just resolves the nested conditionals without branching back:

```
[[ ... ?[ ... ?[ ... ?[ ... ]]?
```

The purpose of the word]]? is to resolve the appropriate number of unresolved forward references since the occurrence of the opening mark [[. There is no exact equivalence for the "andif conditional" in standard Forth, but the closest functionality is the word ENDCASE from Eaker. In the form above, one might invent a word THENS or ENDALL . Note that the ? symbol used in either the]? or the]]? form implies that there is no looping back, merely a termination of a set of conditionals. It should be noted that the word][can be inserted after any use of ?[in the above examples. For simple looping, we use the form:

```
[[ ... ?]
```

to replace the words BEGIN and UNTIL .

Although case statements are not strictly required, we found them useful enough to be included in the basic set. By adding only one additional word we found that we could implement the entire Eaker case statement. The suggested new word is =?[and it has the effect of the sequence

```
OVER = ?[ DROP
```

A case statement typically has the form:

```
[[          n1 =?[ ... ][
            n2 =?[ ... ][
            n3 =?[ ... ][
            default   ]]?
```

We see that we have managed to use previous words for three out of the four words generally used in a case statement. Notice that in the default part it is likely that you will have to DROP the initial argument on the stack.

Finally we come to the counting type of loop. Other than a simple name change, we suggest that Charles Moore's FOR - NEXT loop should replace the older DO - LOOP structure. The simplicity it offers more than pays for the apparent slight loss of generality. The form it takes in LaForth is simply:

```
n #[ ... ]#
```

The appearance of the # sign implies a counting type of a loop, and the appearance is quite similar to ?[..]? in terms of a nice typograghical clustering.

The above information is summarized in the following table for a quick summary:

```
Construct           Usage

Conditional     ?[   ...   ][   ...   ]?
                ?[   ...   ]?
Infinite Loop   [[   ...   ]]
Until Loop      [[   ...   ?]
While Loop      [[   ...   ?[   ...   ?[   ...   ]]
Andif           [[   ...   ?[   ...   ?[   ...   ]]?
Case            [[   ...   =?[   ...   ][
                     ...   =?[   ...   ][
                     ...                ]]?
Down-count      #[   ...   ]#
```

The final table gives suggested pronunciations and the equivalent names of LaForth conditionals as represented in alternative Forth systems:

```
LaForth   Pronunciation     Equivalent          Alternative
[[        mark              CASE                BEGIN
=?[       case              OF                  OVER = IF DROP
?[        ifso              IF                  WHILE
][        otherwise         ENDOF               ELSE
]]?       endall            ENDCASE             THEN THEN ... THEN
]]        repeat            AGAIN               REPEAT
]?        endif             THEN
?]        until             UNTIL
#[        count             FOR
]#        down              NEXT
```

We have presented a more consistent nomenclature for looping and conditional structures than is seen in usual Forth systems. We have ignored some of the obvious additional forms to keep the presentation to a minimum. In the While and Andif forms, additional structures using][and =?[are possible, and in the Case structures one may readily add the simpler conditional structures.

1.3.4. 32 THREADS HASED DICTIONARY

LaForth needs a very fast mechanism to find words in the dictionary so that words can be found and executed instantaneously when a space or carriage return is detected. The dictionary is broken down into 32 different threads, and generally only one thread is traversed to locate a particular word in the dictionary. The link addresses of the last words in the threads are kept in a table, whose address is returned by the word VOCTAB. Words are linked through the link fields in the words. The link field of the first word in a thread contains a zero, indicating the end of the thread.

LaForth also allows for 32 vocabularies to be declared. Each vocabulary is assigned a vocabulary index. The ROOT vocabulary has an index of 0, and the ASSEMBLER vocabulary has an index of 1. Other vocabularies as assigned indices sequentially. The hashing algorithm is very simple. The ASCII value of the first character in a name is added to the index of the vocabulary in which the word belongs. The resulting sum, modulo 32, is used to select the thread in VOCTAB, and the search begins with the link address thus selected from the VOCTAB table.

Each thread links together about 20 words. Searching a word in a thread is thus very fast.

1.3.5. UNIQUE DATA STRUCTURE IN A WORD

Words in LaForth is represented by a single address, the Code Field Address, which is returned by the dictionary search word ' , and is also compiled into other colon definitions. It is also the address which EXECUTE expects.

LaForth uses the Direct Threaded Code technique to organized the information in the Code Field. What it means is that the Code Field contains executable object code, rather than a pointer to a memory location where executable code is stored. The latter method is used in most Forth systems and is known as Indirect Threaded Code. Direct Threaded Code Forth is generally faster, because of the elimination of an extra level of indirection.

In a colon definition, the first three byte in the Code field contain a CALL DOLIST instruction, which invokes the colon definition interpreter. The address list of the colon definition starts from the fourth byte and is terminated by a UNNEST word.

The Link Field in a word is placed two bytes in front of the Code Field, and its contains the address of the link field of the word defined before this word, in the same thread. The link fields thus link together the words in a thread as a linear chain. The address of the link field of the last word in a thread is stored in the vocabulary table VOCTAB. The end of a thread is recognized as the first word has a zero in its link field.

The most outstanding feature of LaForth is the name fields in the words. The Name Field in a word is a variable length field, starting from the 3rd byte below the Code Field and extends to the lower memory. The characters in the name is thus arranged backwards, from high memory address to low memory address. The name field is terminated by a zero byte or NUL character. The null terminated name field allows LaForth to use names of indefinite length, not limited to 31 characters, as in the figForth Model.

The highest bit, bit 7, in the first character in the name field is reserved as the immediate bit. If this bit is set, the word is an immediate word which is executed inside a colon definition. Non-immediate words are compiled in a colon definition. Immediate words are used to construct control and data structures inside a colon definition.