# Forth - The Early Years

Chuck Moore
chipchuck@colorforth.com
1991

## Abstract

Forth is a simple, natural computer language. It has achieved remarkable acceptance where efficiency is valued. It evolved in the 1960s on a journey from university through business to laboratory. This is the story of how a simple interpreter expanded its abilities to become a complete programming language/operating system.

## Forward 1999

This paper was written for the HOPL II (History of programming languages) conference. It was summarily rejected, apparently because of its style. Much of the content was included in the accepted paper [Rather 1993].

This HTML version was reformatted from the original typescript. Minimal changes were made to the text. Examples of source code were suggested by reviewer Phil Koopman. They've not yet been added.

## Contents

## Forth

Forth evolved during the decade of the 60s, across America, within university, business and laboratory, amongst established languages. During this period, I was its only programmer and it had no name until the end. This account is retrieved from memory, prompted by sparse documentation and surviving listings.

Forth is hardly original, but it is a unique combination of ingredients. I'm grateful to the people and organizations who permitted me to develop it - often unbeknownst to them. And to you, for being interested enough to read about it.

Forth is a simple, natural computer language. Today it is accepted as a world-class programming language. That it has achieved this without industry, university or government support is a tribute to its efficiency, reliability and versatility. Forth is the language of choice when its efficiency outweighs the popularity of other languages. This is more often the case in real-world applications such as control and communication.

A number of Forth organizations and a plethora of small companies provide systems, applications and documentation. Annual conferences are held in North America, Europe and Asia. A draft ANSI standard will soon be submitted [ANS 1991].

None of the books about Forth quite capture its flavor. I think the best is still the first, Starting Forth by Leo Brodie [Brodie 1981]. Another window is provided by JFAR's invaluable subject and author index [Martin 1987].

The classic Forth we are discussing provides the minimum support a programmer needs to develop a language optimized for his application. It is intended for a work-station environment: keyboard, display, computer and disk.

Forth is a text-based language that is essentially context-free. It combines 'words' separated by spaces, to construct new words. About 150 such words constitute a system that provides (with date of introduction)

```
      SAO  1958  Interpreter
     SLAC  1961  Data stack
      RSI  1966  Keyboard input
                 Display output, OK
                 Editor
  Mohasco  1968  Compiler
                 Return stack
                 Dictionary
                 Virtual memory (disk)
                 Multiprogrammer
     NRAO  1971  Threaded code
                 Fixed-point arithmetic
```

Such a system has 3-8K bytes of code compiled from 10-20 pages of source. It can easily be implemented by a single programmer on a small computer.

This account necessarily follows my career. But it is intended to be the autobiography of Forth. I will discuss the features listed above; and the names of the words associated with them. The meaning of many words is obvious. Some warrant description and some are beyond the scope of this paper.

That portion of the Forth dictionary to be mentioned is summarized here:

```
Interpreter
     WORD  NUMBER  INTERPRET  ABORT
     HASH  FIND  '  FORGET
     BASE  OCTAL  DECIMAL  HEX
     LOAD  EXIT  EXECUTE  (
Terminal
```

```
     KEY  EXPECT
     EMIT  CR  SPACE  SPACES  DIGIT  TYPE  DUMP
Data stack
     DUP  DROP  SWAP  OVER
     +  -  *  /  MOD  NEGATE
     ABS  MAX  MIN
     AND  OR  XOR  NOT
     0<  0=  =
     @  !  +!  C@  C!
     SQRT  SIN.COS  ATAN  EXP  LOG
Return stack
     :  ;  PUSH  POP  I
Disk
     BLOCK  UPDATE  FLUSH  BUFFER  PREV  OLDEST
Compiler
     CREATE  ALLOT  ,  SMUDGE
     VARIABLE  CONSTANT
     [  ]  LITERAL  ."  COMPILE
     BEGIN  UNTIL  AGAIN  WHILE  REPEAT
     DO  LOOP  +LOOP  IF  ELSE  THEN
```

## MIT, SAO, 1958

October, 1957 was Sputnik - a most exciting time. I was a sophmore at MIT and got a part-time job with SAO (Smithsonian Astrophysical Observatory, 14 syllables) at Harvard.

SAO was responsible for optical tracking of satellites - Moonwatch visual observations and Baker-Nunn tracking cameras. Caught off-guard by Sputnik, they hired undergraduates to compute predictions with Friden desk calculators. John Gaustad told me about MIT's IBM EDPM 704 and loaned me his Fortran II manual. My first program, Ephemeris 4, eliminated my job [Moore 1958].

Now a Programmer, I worked with George Veis to apply his method of least-squares fitting to determine orbital elements, station positions and ultimately the shape of Earth [Veis 1960]. Of course, this part-time job was at least 40 hours, and yes, my grades went to hell.

At MIT, John McCarthy taught an incredible course on LISP. That was my introduction to recursion, and to the marvelous variety of computer language. Wil Baden has noted that LISP is to Lambda Calculus as Forth is to Lukasewcleicz Postfix.

APL was also a topical language, with its weird right-left parsing. Although I admire and emulate its operators, I'm not persuaded they constitute an optimal set.

The programming environment in the 50s was more severe than today. My source code filled 2 trays with punch cards. They had to be carried about to be put through machines, mostly by me. Compile took 30 minutes (just like C) but limited computer time meant one run per day, except maybe 3rd shift.

So I wrote this simple interpreter to read input cards and control the program. It also directed calculations. The five orbital elements each had an empirical equation to account for atmospheric drag and the non-spherical Earth. Thus I could compose different equations for the several satellites without re-compiling.

These equations summed terms such as P2 (polynomial of degree 2) and S (sine). 36-bit floating-point dominated calculation time so overhead was small. A data stack was unnecessary, and probably unknown to me.

The Forth interpreter began here with the words

```
WORD  NUMBER  INTERPRET  ABORT
```

They weren't spelled that way because they were statement numbers.

INTERPRET uses WORD to read words separated by spaces and NUMBER to convert a word to binary (in this case, floating-point). Such free-format input was unusual, but was more efficient (smaller and faster) and reliable. Fortran input was formatted into specific columns and typographic errors had caused numerous delays.

This interpreter used an IF ... ELSE IF construct, coded in Fortran, finding a match on a single character. Error handling consisted of terminating the run. Then, as now, ABORT asked the user what to do. Since input cards were listed as they were read, you knew where the error was.

## Stanford, SLAC, 1961

In 1961 I went to Stanford to study mathematics. Although Stanford was building its computer science department, I was interested in real computing. I was impressed that they could (dared?) write their own Algol compiler. And I fatefully encountered the Burroughs B5500 computer.

I got another 'part-time' job at SLAC (Stanford Linear Accelerator Center, 12 syllables) writing code to optimize beam steering for the pending 2-mile electron accelerator. This was a natural application of my least-squares experience to phase-space. Hal Butler was in charge of our group and the program, TRANSPORT, was quite successful.

Another application of least-squares was the program CURVE, coded in Algol (1964). It is a general-purpose non-linear differential-corrections data-fitting program. Its statistical rigor provides insight into agreement between model and data.

The data format and model equations were interpreted and a push-down stack used to facilitate evaluation. CURVE was an impressive precursor to Forth. It introduced these words to provide the capability to fit models much more elaborate than simple equations:

```
+  -  *  NEGATE
IF  ELSE  THEN  <
DUP  DROP  SWAP
:  ;  VARIABLE  !  (
SIN  ATAN  EXP  LOG
```

Spelling was quite different:

```
NEGATE  was  MINUS
```

```
     DROP       ;
     SWAP       .
        !       <
 VARIABLE       DECLARE
        ;       END
   ( ...)       COMMENT ...;
```

The interpreter used IF ... ELSE IF to identify a 6-character input word called ATOM (from LISP). DUP DROP and SWAP are 5500 instructions; I'm surprised at the spelling change. The word : was taken from the Algol label format, flipped for left-right parsing (to prevent the interpreter encountering an undefined word):

```
Algol - LABEL:
CURVE - : LABEL
```

In fact, : marked a position in the input string to be interpreted later. Interpretation was stopped by ; . A version of : was named DEFINE .

The store operator ( ! ) appeared in connection with VARIABLE . But fetching ( @ ) was automatic. Note the input had become complex enough to warrant comments. The sometime-criticised postfix conditional dates from here:

```
Algol - IF expression THEN true ELSE false
CURVE - stack IF true ELSE false THEN
```

True is interpreted if stack is non-zero. THEN provides unique termination, the lack of which always confused me in Algol. Such expressions were interpreted: IF would scan ahead for ELSE or THEN.

The word < introduces the convention that relations leave a truth value on the stack, 1 for true and 0 for false. The transcendental functions are, of course, library calls.

## Free-lance

I left Stanford in 1965 to become a free-lance programmer in the New York City area. This was not unusual, and I found work programming in Fortran, Algol, Jovial, PL/I and various assemblers. I literally carried my card deck about and recoded it as necessary.

Minicomputers were appearing, and with them terminals. The interpreter was ideal for teletype input, and soon included code to handle output. So we aquire the words

```
KEY  EXPECT
EMIT  CR  SPACE  SPACES  DIGIT  TYPE
```

EXPECT is a loop calling KEY to read a keystroke. TYPE is a loop calling EMIT to display a character.

With the TTY came paper-tape and some of the most un-friendly software imaginable - hours of editing and punching and loading and assembling and printing and loading and testing and repeating. I remember a terrible Sunday in a Manhattan skyscraper when I couldn't find splicing tape (nothing else works) and swore that 'There must be a better way'.

I did considerable work for Bob Davis at Realtime Systems, Inc (RSI). I became a 5500 MCP guru to support his time-sharing service (remote input to a mainframe) and wrote a Fortran-Algol translator and file editing utilities. The translator taught me the value of spaces between words, not required by Fortran.

The interpreter still accepted words with the first 6 characters significant (the 5500 had 48-bit words). The words

```
LIST  EDIT  BEGIN  AGAIN  EXIT
```

appear, with BEGIN ... AGAIN spelled START ... REPEAT and used to bracket the editor commands

```
T  TYPE  I  INSERT  D  DELETE  F  FIND
```

later used in NRAO's editor. The word FIELD was used in the manner of Mohasco and Forth, Inc's data-base management.

One of Forth's distinctive features comes from here. The rule is that Forth acknowledge each line of input by appending OK when interpretation is complete. This may be difficult, for when input is terminated by CR a blank must be echoed, and the CR included with OK. At RSI, OK was on the next line, but it still conveyed friendly reassurance over an intimidating communications line:

```
56 INSERT ALGOL IS VERY ADAPTABLE
OK
```

This postfix notation suggests a data stack, but it only had to be one deep.

## Mohasco, 1968

In 1968 I transformed into a business programmer at Mohasco Industries, Inc in Amsterdam NY. They are a major home-furnishing company - carpets and furniture. I had worked with Geoff Leach at RSI and he persuaded me to follow him up-state. I had just married, and Amsterdam has a lovely small-town atmosphere to contrast with NYC.

I rewrote my code in COBOL and learned the truth about business software. Bob Rayco was in charge of Corporate data processing and assigned me two relevant projects:

He leased an IBM 1130 minicomputer with a 2250 graphic display. The object was to see if computer graphics helped design patterned carpets. The answer was 'not without color' and the 1130 went away.

Meanwhile I had the latest minicomputer environment: 16-bit CPU, 8K RAM, disk (my first), keyboard, printer, card reader/punch, Fortran compiler. The reader/punch provided disk backup. I ported my interpreter again (back to Fortran) and added a cross-assembler to generate code for the

2250.

The system was a great success. It could draw animated 3-D images when IBM could barely draw static 2-D. Since this was my first real-time graphics, I coded Spacewar, that first video game. I also converted my Algol chess program into Forth and was duely impressed how much simpler it became.

The file holding the interpreter was labeled FORTH, for 4th (next) generation software - but the operating system restricted file names to 5 characters.

This environment for programming the 2250 was far superior to the Fortran environment, so I extended the 2250 cross-assembler into an 1130 compiler. This introduced a flock of words

```
DO  LOOP  UNTIL
BLOCK  LOAD  UPDATE  FLUSH
BASE  CONTEXT  STATE  INTERPRET  DUMP
CREATE  CODE  ;CODE  CONSTANT  SMUDGE
@  OVER  AND  OR  NOT  0=  0<
```

They were still differently spelled

| | | |
|---|---|---|
| LOOP | was | CONTINUE |
| UNTIL | | END |
| BLOCK | | GET |
| LOAD | | READ |
| TYPE | | SEND |
| INTERPRET | | QUERY |
| CREATE | | ENTER |
| CODE | | the cent symbol |

The only use I've ever found for the cent symbol. The loop index and limit were on the data stack. DO and CONTINUE were meant to acknowledge Fortran.

BLOCK manages a number of buffers to minimize disk access. LOAD reads source from a 1024-byte block. 1024 was chosen as a nice modular amount of disk, and has proven a good choice. UPDATE allows a block to be marked and later rewritten to disk, when its buffer is needed (or by FLUSH ). It implements virtual memory and is concealed in store ( ! ) words.

BASE allows octal and hex numbers as well as decimal. CONTEXT was the first hint of vocabularies and served to isolate editor words. STATE distinguished compiling from interpreting. During compilation, the count and first 3 characters of a word were compiled for later interpretation. Strangely, words could be terminated by a special character, an aberration quickly abandoned. The fetch operator ( @ ) appeared in many guises, since fetching from variables, arrays and disk had to be distinguished. DUMP became important for examining memory.

But most important, there was now a dictionary. Interpret code now had a name and searched a linked-list for a match. CREATE constructs the classic dictionary entry:

```
link to previous entry
count and 3 characters
```

```
code to be executed
parameters
```

The code field was an important innovation, since an indirect jump was the only overhead, once a word had been found. The value of the count in distinguishing words, I learned from the compiler writers of Stanford.

An important class of words appeared with CODE . Machine instructions followed in the parameter field. So any word within the capability of the computer could now be defined. ;CODE specifies the code to be executed for a new class of words, and introduced what are now called objects.

SMUDGE avoided recursion during the interpretation of a definition. Since the dictionary would be searched from newest to oldest definitions, recursion would normally occur.

Finally, the return stack appeared. Heretofor, definitions had not been nested, or used the data stack for their return address. Altogether a time of great innovation in the punctuated evolution of Forth.

The first paper on Forth, an internal Mohasco report, was written by Geoff and me [Moore 1970]. It would not be out of place today.

In 1970 Bob ordered a Univac 1108. An ambitious project to support a network of leased lines for an order-entry system. I had coded a report generator in Forth and was confident I could code order-entry. I ported Forth to the 5500 (standalone!) to add credibility. But corporate software was COBOL. The marvelous compromise was to install a Forth system on the 1108 that interfaced with COBOL modules to do transaction processing.

I vividly recall commuting to Schenectady that winter to borrow 1107 time 3rd shift. My TR4-A lacked floor and window so it became a nightly survival exercise. But the system was an incredible success. Even Univac was impressed with its efficiency (Les Sharp was project liason). The ultimate measure was response time, but I was determined to keep it maintainable (small and simple). Alas, an economic downturn led Management to cancel the 1108. I still think it was a bad call. I was the first to resign.

1108 Forth must have been coded in assembler. It buffered input and output messages and shared the CPU among tasks handling each line. Your classic operating system. But it also interpreted the input and PERFORMed the appropriate COBOL module. It maintained drum buffers and packed/unpacked records. The words

```
BUFFER  PREV  OLDEST
TASK  ACTIVATE  GET  RELEASE
```

date from here. BUFFER avoided a disk read when the desired block was known empty. PREV (previous) and OLDEST are system variables that implement least-recently-used buffer management. TASK defines a task at boot time and ACTIVATE starts it when needed. GET and RELEASE manage shared resources (drum, printer). PAUSE is how a task relinquishes control of the CPU. It is included in all I/O operations and is invisible to transaction code. It allows a simple round-robin scheduling algorithm that avoids lock-out.

After giving notice, I wrote an angry poem and a book that has never been published. It described how to develop Forth software and encouraged simplicity and innovation. It also described indirect-threaded code, but the first implementation was at NRAO.

I struggled with the concept of meta-language, language that talks about language. Forth could now interpret an assembler, that was assembling a compiler, that would compile the interpreter. Eventually I decided the terminology wasn't helpful, but the term Meta-compile for recompiling Forth is still used.

## NRAO, 1971

George Conant offered me a position at NRAO (National Radio Astronomy Observatory, 15 syllables). I had known him at SAO and he liked Ephemeris 4. So we moved to Charlottesville VA and spent summers in Tucson AZ when the radio-telescope on Kitt Peak was available for maintainance.

The project was to program a Honeywell 316 minicomputer to control a new filter-bank for the 36' millimeter telescope. It had a 9-track tape and Tektronix storage-tube terminal. George gave me a free hand to develop the system, though he wasn't pleased with the result. NRAO was a Fortran shop and by now I was calling Forth a language. He was right in that organizations have to standardize on a single language. Other programmers now wanted their own languages.

Anyhow, I had coded Forth in assembler on the IBM 360/50 mainframe. Then I cross-compiled it onto the 316. Then I re-compiled it on the 316 (Although I had a terminal on the 360, response time was terrible). The application was easy once the system was available. There were two modes of observing, continuum and spectral-line. Spectral-line was the most fun, for I could display spectra as they were collected and fit line-shapes with least-squares [Moore 1973].

The system was well-received in Tucson, where Ned Conklin was in charge. It did advance the state-of-the-art in on-line data reduction. Astronomers used it to discover and map inter-stellar molecules just as that became hot research.

Bess Rather was hired to provide on-site support. She had first to learn the Forth system and then explain and document it, with minimal help from me. The next year I reprogrammed the DDP-116 to optimize telescope pointing. The next, Bess and I replaced the 116 and 316 with a DEC PDP-11.

The development that made all this possible was indirect-threaded code. It was a natural development from my work at Mohasco, though I later heard that DEC had used direct-threaded code in one of their compilers. Rather than re-interpret the text of a definition, compile the address of each dictionary entry. This improved efficiency for each reference required only 2 bytes and an address interpreter could sequence through a definition enormously faster. In fact, this interpreter was a 2-word macro on the 11:

```
    : NEXT   IP )+ W MOV  W )+ ) JMP ;
```

Now Forth was complete. And I knew it. I could write code more quickly that was more efficient and reliable. Moreover, it was portable. I proceeded to recode the 116 pointing the 300' Green Bank telescope, and the HP mini that was inaugurating VLBI astronomy. George gave me a ModComp and I did Fourier transforms for interferometry and pulsar search (64K data). I even demonstrated that complex multiply on the 360 was 20% faster in Forth than assembler.

NRAO appreciated what I had wrought. They had an arrangement with a consulting firm to identify spin-off technology. The issue of patenting Forth was discussed at length. But since software patents were controversial and might involve the Supreme Court, NRAO declined to pursue the matter. Whereupon, rights reverted to me. I don't think ideas should be patentable. Hindsight agrees that Forth's only chance lay in the public domain. Where it has flourished.

Threaded-code changed the structure words (such as DO LOOP IF THEN ). They acquired an elegant implementation with addresses on the data stack during compilation.

Now each Forth had an assembler for its particular computer. It uses post-fix op-codes and composes addresses on the data stack, with Forth-like structure words for branching. The manufacturer's mnemonics are defined as word classes by ;CODE . Might take an afternoon to code. An example is the macro for NEXT above.

Unconventional arithmetic operators proved their value

```
M*  */  /MOD  SQRT  SIN.COS  ATAN  EXP  LOG
```

M* is the usual hardware multiply of 2 16-bit numbers to a 32-bit product (arguments, of course, on the data stack). */ follows that with a divide to implement rational arithmetic. /MOD returns both quotient and remainder and is ideal for locating records within a file. SQRT produces a 16-bit result from a 32-bit argument. SIN.COS returns both sine and cosine as is useful for vector and complex arithmetic (FFT). ATAN is its inverse and has no quadrant ambiguity. EXP and LOG were base 2.

These functions used fixed-point arithmetic - 14 or 30 bits right of a binary point for trig, 10 for logs. This became a characteristic of Forth since it's simpler, faster and more accurate than floating-point. But hardware and software floating-point are easy to implement.

I'd like to applaud the invaluable work of Hart [Hart 1978] in tabulating function approximations with various accuracies. They have provided freedom from the limitations of existing libraries to those of us in the trenches.

The word DOES> appeared (spelled ;: ). It defines a class of words (like ;CODE ) by specifying the definition to be interpreted when the word is referenced. It was tricky to invent, but particularly useful for defining op-codes.

Nonetheless, I failed to persuade Charlottesville that Forth was suitable. I wasn't going to be allowed to program the VLA. Of any group, 25% like Forth and 25% hate it. Arguments can get violent and compromise is rare. So the friendlies joined forces and formed Forth, Inc. And that's another story.

## Moral

The Forth story has the making of a morality play: Persistant young programmer struggles against indifference to discover Truth and save his suffering comrades. It gets better: Watch Forth. Inc go head to head with IBM over a French banking system.

I know Forth is the best language so far. I'm pleased at its success, especially in the ultra-conservative arena of Artificial Intelligence. I'm disturbed that people who should, don't appreciate how it embodies their own description of the ideal programming language.

But I'm still exploring without license. Forth has led to an architecture that promises a wonderful integration of software and silicon. And another new programming environment.

## References

[ANS 1991] Draft Proposed ANS Forth, document number X3.215-199x, available from Global Engineering Documents, 2805 McGaw Ave., Irvine CA 92714.

[Brodie, 1981] Brodie, Leo, Starting FORTH, Englewood Cliffs NJ: Prentice-Hall, 1981, ISBN 0 13 842930 8.

[Hart, 1968] Hart, John F. et al, Computer Approximations. Malabar FL: Krieger, 1968; (Second Edition), 1978, ISBN 0 88275 642 7.

[Martin, 1987] Martin, Thea, A Bibliography of Forth References, 3rd Ed, Rochester NY: Institute for Applied Forth Research, 1987, ISBN 0 914593 07 2.

[Moore, 1958] Moore, Charles H. and Lautman, Don A., Predictions for photographic tracking stations - APO Ephemeris 4, in SAO Special Report No. 11, Schilling G. F., Ed., Cambridge MA: Smithsonian Astrophysical Observatory, 1958 March.

[Moore, 1970] --- and Leach, Geoffrey C., FORTH - A Language for Interactive Computing, Amsterdam NY: Mohasco Industries, Inc. (internal pub.) 1970.

[Moore, 1972] --- and Rather, Elizabeth D., The FORTH program for spectral line observing on NRAO's 36 ft telescope, Astronomy & Astrophysics Supplement Series, Vol. 15, No. 3, 1974 June, Proceedings of the Symposium on the Collection and Analysis of Astrophysical Data, Charlottesville VA, 1972 Nov. 13-15.

[Moore, 1980] ---, The evolution of FORTH, an unusual language, Byte, 5:8, 1980 August.

[Rather, 1993] Rather, Elizabeth D., Colburn, Donald R. and Moore, Charles H., The Evolution of Forth, in History of Programming Languages-II, Bergin T. J. and Gibson, R. G., Ed., New York NY: Addison-Wesley, 1996, ISBN 0-201-89502-1.

[Veis, 1960] Veis, George and Moore, C. H., SAO differential orbit improvement program, in Tracking Programs and Orbit Determination Seminar Proceedings, Pasadena CA: JPL, 1960 February 23-26.