(What may now seem) syntax oddities in older programming languages

Notes to accompany a talk given to CamPUG

The slides, and this document itself, can be found at https://github.com/tibs/old-proglang-syntaxes-talk

> **Note** that this is being rewritten to make sense as an article, not just my own random notes. Progress so far:
>
> - FORTRAN IV rewritten
> - LISP rewritten
> - COBOL rewritten
> - Forth started / in progress

Contents

Introduction

The intent of these notes is to show that language design is not obvious. Those writing early programming languages had to design their languages with relatively little prior art (indeed, in some cases, with no prior art). This could lead to some choices that seem odd to us now. Sometimes those choices were led by technical (hardware) limitations, and sometimes they were an experiment.

It is also worth remembering that at the time, designing and implementing a new programming language was a very large task. Certainly at the beginning, the parsing techniques we are now used to did not exist. There wasn't even a standard syntax for describing a programming language syntax.

All of the languages, except a couple of exceptions toward the end, are ones I knew about in the 1980s. I've chosen these particular languages because I have some familiarity with them, or because I consider them fascinating for one reason or another.

There are compilers or interpreters available for more-or-less all of them still available, and in a good many cases, the language (or its direct descendant or descendants) is still in use.

There are, of course, many other choices I could have made. If you're at all interested, I give some links at the end that can start you off on your own investigations.

FORTRAN IV

https://en.wikipedia.org/wiki/Fortran

FORTRAN is arguably the first high level programming language, and it is definitely the oldest high level language still in use today, although of course it has changed a lot over the years.

The first FORTRAN compiler was in 1957, and FORTRAN IV was released in 1962.

I've chosen to look at FORTRAN IV (rather than any other version of FORTRAN) because it's the first programming language I had to learn at university, and because we were still using it in the early 1980s at my first job. (Incidentally, the FORTRAN 66 standard essentially describes the same language, but no-one calls it that.)

Note that modern Fortran is no longer very much like this - it's continued to evolve (this is also true for LISP and COBOL, the other two early languages that are still in wide use, which we shall come to later).

Here is an example of fairly traditional FORTRAN IV, implementing the "99 Bottles" song:

```
      INTEGER BOTTLS
      DO 50 I = 1, 99
        BOTTLS = 100 - I
        PRINT 10, BOTTLS
        PRINT 20, BOTTLS
        PRINT 30
        BOTTLS = BOTTLS - 1
        PRINT 10, BOTTLS
        PRINT 40
50    CONTINUE
      STOP
10    FORMAT(1X, I2, 31H bottle(s) of beer on the wall.)
20    FORMAT(1X, I2, 19H bottle(s) of beer.)
30    FORMAT(34H Take one down and pass it around,)
40    FORMAT(1X)
      END
```

(based on the example at http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/f.html#FORTRAN-IV by Allen Mcintosh, mcintosh@bellcore.com, but changed to upper case and re-ordered to follow a different coding style.)

Punched cards

The syntax (and particularly the layout) of FORTRAN [1] was heavily influenced by its input medium, punched cards:

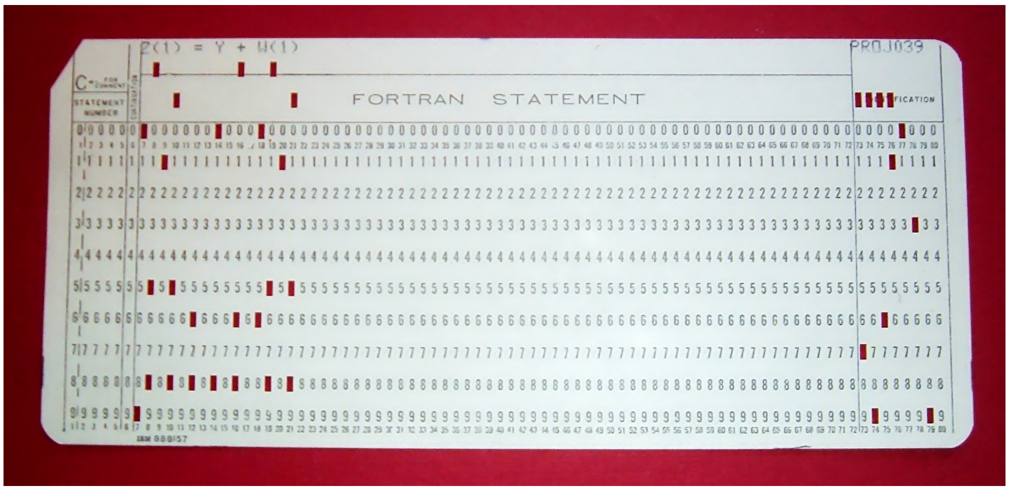


image source: https://en.wikipedia.org/wiki/Computer_programming_in_the_punched_card_era

The above is a punched card representing the FORTRAN line of code:

```
Z(1) = Y + W(1)
```

(I'm assuming that from the text at the top of the card - I haven't actually decoded the hole sequences.)

There are three sorts of card used by FORTRAN: **Data cards**, **comment cards** and **statement cards**.

Data cards are used for the input and output of data. All 80 columns are used to indicate characters.

Comment cards have a "C" in the first column. The rest of the characters on the card are ignored by the compiler.

Statement cards represent a single "line" of FORTRAN code.

Statement cards have four sections:

```
                  1         2           3         4         5         6         7            8
12345 | 6 | 789012345678901234567890123456789012345678901234567890123456789012 | 34567890
```

- The first five characters can be unique statement numbers. These are used to refer to the statement from elsewhere in the program. They don't have to occur in a particular order.
- The sixth character may be a "continuation" character. If it is present and not "0", then this card is a continuation of the previous card.

  (So for a continued line, one might put a "0" in that column for the first card, a "1" for the second card, and so on. I don't remember if we followed that convention ourselves.)
- Positions 7 - 72 are used for the actual program code.

  In our example card, this is `Z(1) = Y + W(1)`
- Positions 73 - 80 are ignored by the compiler, but would typically be used as a sequence indicator. This is useful for indicating the order of the cards (we actually had this as an exercise at University - we were given a deck of FORTRAN punched cards, without sequence numbers, and told that "they had been dropped" and we were to put them back into order.)

  In our example card, this is `PROJ039` (or so the text at the top suggests).

On a punched card, columns 1 - 6 could be left unpunched if there was no statement number or continuation character, but when typing FORTRAN code into a text editor, actual spaces would be used (use of tabs is beyond the scope of this article).

Here is that earlier example with • characters replacing leading spaces, to mak the layout a bit more obvious:

```
••••••INTEGER BOTTLS
••••••DO 50 I = 1, 99
••••••••BOTTLS = 100 - I
••••••••PRINT 10, BOTTLS
••••••••PRINT 20, BOTTLS
••••••••PRINT 30
••••••••BOTTLS = BOTTLS - 1
••••••••PRINT 10, BOTTLS
••••••••PRINT 40
50••••CONTINUE
••••••STOP
10••••FORMAT(1X, I2, 31H bottle(s) of beer on the wall.)
20••••FORMAT(1X, I2, 19H bottle(s) of beer.)
30••••FORMAT(34H Take one down and pass it around,)
40••••FORMAT(1X)
••••••END
```

Some notes on FORTRAN IV syntax

Spaces are not significant

More precisely, spaces in the program code are ignored. So `GOTO 99` is the same as `GO TO 99` and also the same as `G O T O 9 9`.

There are no reserved words

Statements like:

```
IF (IF.EQ.THEN) IF=IF*THEN
```

are perfectly sensible (although perhap ill-advised).

Six character names

Names were restricted to 6 characters (hence `BOTTLS = 99`). This made writing libraries interesting. We would typically name library functions using 3 letters as a mnemonic for the library, and then 3 characters to identify what the function was.

No If/then/else

If/then/else hadn't been invented when FORTRAN IV was defined. The basic IF was of the form:

```
IF (something) expression
```

For instance:

```
IF (VAL.GT.9) VAL = 0

IF (VAL.EQ.3) GOTO 1000
```

The arithmetic IF

```
IF (X/Y*Z) 100,300,50
```

If the result of X/Y*Z is negative, go to statement number 100, if zero go to statement number 300, and if positive go to statement number 50.

This felt very useful at the time, but could quickly lead to spaghetti code.

Implicit typing of variables

You could declare the type of a variable explicitly:

```
INTEGER DAY,WEEK,MONTH
```

but if you did not, then the type would be decided based on the first character of the name:

```
C A variable starting I - N defaults to INTEGER, otherwise REAL
      I = 4
      R = 3.0
```

FUNCTIONS versus SUBROUTINES

A function returns a single value, assigned to the function name. For instance:

```
INTEGER FUNCTION ADD1(I)
  ADD1 = I + 1
END

J = ADD1(3)
```

A subroutine returns 0 or more values, via its argument list. For instance:

```
SUBROUTINE CALC(A,B,C,SUM,SUMSQ)
  SUM = A + B + C
  SUMSQ = SUM ** 2
END

CALL CALC(1,2,3,SUM1,SUMSQ1)
```

LISP

https://en.wikipedia.org/wiki/Lisp_(programming_language))

LISP is one year younger than FORTRAN, which makes it the second oldest programming language still in common use.

LISP was originally specified in 1958.

Modern lisps abound, including Common Lisp and a whole host of Schemes. The only Lisp I have any direct experience of writing myself, though, is Emacs Lisp.

M-expressions and S-expressions

It didn't end up quite how it was initially designed.

From https://en.wikipedia.org/wiki/Lisp_(programming_language))#History:

> McCarthy's original notation used bracketed "M-expressions" that would be translated into S-expressions.
>
> ...

> Once Lisp was implemented, programmers rapidly chose to use S-expressions, and M-expressions were abandoned.

The Lisp 1.5 manual talks about both forms. As an example taken from there, the M-expression:

```
[atom[x] → x; T → ff[car[x]]]
```

corresponds to the S-expression:

```
(COND ((ATOM X) X)
    ((QUOTE T) (FF (CAR X))))
```

From https://en.wikipedia.org/wiki/M-expression:

> McCarthy had planned to develop an automatic Lisp compiler (LISP 2) using M-expressions as the language syntax and S-expressions to describe the compiler's internal processes. Stephen B. Russell read the paper and suggested to him that S-expressions were a more convenient syntax. Although McCarthy disapproved of the idea, Russell and colleague Daniel J. Edwards hand-coded an interpreter program that could execute S-expressions. This program was adopted by McCarthy's research group, establishing S-expressions as the dominant form of Lisp.

From http://www.softwarepreservation.org/projects/LISP/lisp2/SP-2450-SUMSQUARE_LCS.pdf we have a LISP 2 M-expression:

```
% SUMSQUARE COMPUTES THE SUM OF THE SQUARES OF THE
% COMPONENTS OF AN ARBITRARY VECTOR

REAL SECTION COMPUTE, LISP;

REAL FUNCTION SUMSQUARE(X(I));
   BEGIN INTEGER J; REAL Y;
           FOR J ← STEP 1 UNTIL I DO
               Y ← Y + X(J) ↑ 2;
           RETURN Y;
   END;

SUMSQUARE (2, 7, 4); STOP
```

giving the result:

```
69.0
```

In Common Lisp this might be written:

```
(defun sum-of-squares (vector)
  (loop for x across vector sum (expt x 2)))
```

(source from https://rosettacode.org/wiki/Sum_of_squares#Common_Lisp)

or in Scheme:

```
(define (sum-of-squares l)
  (apply + (map * l l)))
```

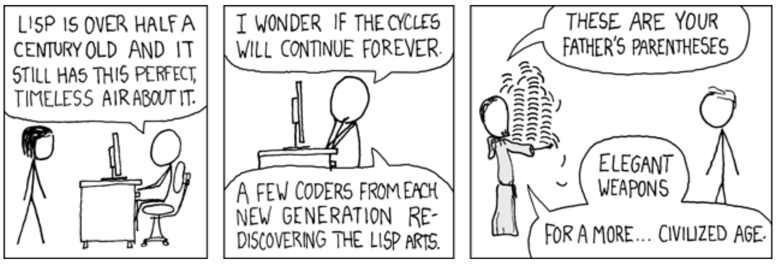
(source https://rosettacode.org/wiki/Sum_of_squares#Scheme)

The big advantage of S-expressions is that they act as both a data representation and a program representation, which means that it is possible to treat a Lisp program itself as data in a very natural manner.

My father's parentheses

It is definitely true that many people find Lisp daunting.

I think that is in large part because of the parentheses:


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

https://xkcd.com/297/ (Randall Monroe)

but probably also that Lisp is relatively unusual in using prefix notation (`(+ 1 2)` rather than `1 + 2`).

A good text editor should handle indentation and balancing the parentheses for you, but it is true that Lisp is visually unlike other programming languages.

Interestingly, Franz Lisp recognised the problem of sometimes needing to type many closing parentheses in sequence, and allowed the use of `]` to mean "close all outstanding `)`". I'm not sure how much that feature was used.

LISP - as we know it

Some more 99 Bottles examples:

Scheme, from http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/s.html#Scheme

```
;;; Tim Goodwin (tim@pipex.net)

(define bottles
  (lambda (n)
    (cond ((= n 0) (display "No more bottles"))
          ((= n 1) (display "One bottle"))
          (else (display n) (display " bottles")))
    (display " of beer")))

(define beer
  (lambda (n)
    (if (> n 0)
        (begin
          (bottles n) (display " on the wall") (newline)
          (bottles n) (newline)
          (display "Take one down, pass it around") (newline)
          (bottles (- n 1)) (display " on the wall") (newline)
          (newline)
          (beer (- n 1))))))

(beer 99)
```

Common Lisp, from https://rosettacode.org/wiki/99_Bottles_of_Beer/Lisp

```
(defun bottles (x)
  (loop for bottles from x downto 1
        do (format t "~a bottle~:p of beer on the wall~@
                     ~:*~a bottle~:p of beer~@
```

```
                             Take one down, pass it around~@
                             ~V[No more~:;~:*~a bottle~:p of~] beer on the wall~2%"
                        bottles (1- bottles))))

(bottles 99)
```

Racket, from https://rosettacode.org/wiki/99_Bottles_of_Beer#Racket

```
#lang racket
(define (sing bottles)
  (define (plural n) (~a n " bottle" (if (= n 1) "" "s")))
  (printf "~a of beer on the wall\n~a of beer\n~
          Take one down, pass it around\n~a of beer on the wall\n\n"
          (plural bottles) (plural bottles) (plural (sub1 bottles)))
  (unless (= 1 bottles) (sing (sub1 bottles))))
(sing 99)
```

If-then-else

According to https://en.wikipedia.org/wiki/Lisp_(programming_language

> A conditional using an if–then–else syntax was invented by McCarthy in a Fortran context. He proposed its inclusion in ALGOL, but it was not made part of the Algol 58 specification. For Lisp, McCarthy used the more general cond-structure. Algol 60 took up if–then–else and popularized it.

So Algol 60 got "if-then-else" and LISP got `cond`, which looks more like the `case` or `switch` statement we're used to in other programming languages

COBOL

https://en.wikipedia.org/wiki/COBOL

COBOL is the third oldest programming language still in common use. It was designed in 1959 and first standardised in 1968.

Both FORTRAN (FORmula TRANslation) and LISP (LISt Processing) were seen as languages for mathematicians and engineers. A need was seen for a programming language that could be used by non-technical people for business applications. In order to achieve this, COBOL (Common Business Oriented Language) tried to use English words and phrases rather than mathematical notations.

It is possible to regard COBOL as the beginning of a long tradition of trying to make programming more accessible to non-academics/non-programmers. Some programming languages continue the tradition of trying to be "English like", for instance Inform 7 in the text adventure space (this paper from 2019 gives a good introduction to its history) or AppleScript. Others, like Blockly, Scratch and LabVIEW, try using graphical techniques - these could really only become practicable after our period of interest, as graphical hardware became easily available.

COBOL was also the first popular language that was designed to be machine (hardware and operating system) agnostic, which goes well with its aim to be suitable for non-computer experts.

It also introduced sophisticated mechanisms for organising and introspecting data, and for the input and output of that data.

https://en.wikipedia.org/wiki/Visual_programming_language

COBOL programs have a reputation for verbosity. For instance, the 99 Beers example from http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/c.html#Cobol is quite long (and also shows how the language uses significant column layout):

```
IDENTIFICATION DIVISION.
PROGRAM-ID.BOTTLES_OF_BEER.
AUTHOR.DONALD FRASER.
*
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX.
OBJECT-COMPUTER. VAX.
*
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OUTPUT-FILE
                ASSIGN TO BEERS_ON_THE_WALL.
*
DATA DIVISION.
FILE SECTION.
FD OUTPUT-FILE
        LABEL RECORDS ARE OMITTED.
01 BEERS-OUT                                 PIC X(133).
*
WORKING-STORAGE SECTION.
01 FLAGS-COUNTERS-ACCUMULATORS.
        05 FLAGS.
                10 E-O-F                               PIC 9.
                        88 END-OF-FILE               VALUE 1.
        05 COUNTERS.
                10 BOTTLES                   PIC 999
                                             VALUE 0.
01 RECORD-OUT.
        05 LINE1.
                10 NUMBER-OF-BEERS-1                   PIC ZZ9.
                10                                     PIC X(28)
                            VALUE "BOTTLES OF BEER IN THE WALL ".
                10                                                     PIC
X
                            VALUE ",".
                        10 NUMBER-OF-BEERS-2           PIC ZZ9.
                10                                                     PIC
X.
                10                                     PIC X(17)
                            VALUE "BOTTLES OF BEER.".
        05 LINE2.
                10                                             PIC X(34)
                            VALUE "TAKE ONE DOWN AND PASS IT ARROUND ".
                10 NUMBER-OF-BEERS-3         PIC ZZ9.
                10                                     PIC X.
                10                                     PIC X(28)
                            VALUE "BOTTLES OF BEER IN THE WALL".
*
PROCEDURE DIVISION.
DRIVER-MODULE.
      PERFORM INITIALIZATION.
      PERFORM PROCESS UNTIL END-OF-FILE.
      PERFORM TERMINATION.
      STOP RUN.
*
INITIALIZATION.
```

```
         OPEN OUTPUT OUTPUT-FILE.
         ADD 100 TO BOTTLES.
*

PROCESS.
         IF BOTTLES = 0 THEN
                 COMPUTE E-O-F = 1
         ELSE PERFORM WRITE-ROUTINE
         END-IF.
*
TERMINATION.
         CLOSE OUTPUT-FILE.
*
WRITE-ROUTINE.
         MOVE BOTTLES TO NUMBER-OF-BEERS-1, NUMBER-OF-BEERS-2.
         COMPUTE BOTTLES = BOTTLES - 1.
         WRITE BEERS-OUT FROM LINE1.
         MOVE BOTTLES TO NUMBER-OF-BEERS-3.
         WRITE BEERS-OUT FROM LINE2.
```

More modern versions of COBOL are a lot more concise. This example from https://rosettacode.org/wiki/99_Bottles_of_Beer#COBOL (which I think is COBOL 2002) "adheres to the minimum guidelines":

```
program-id. ninety-nine.
data division.
working-storage section.
01  cnt       pic 99.

procedure division.

  perform varying cnt from 99 by -1 until cnt < 1
    display cnt " bottles of beer on the wall"
    display cnt " bottles of beer"
    display "Take one down, pass it around"
    subtract 1 from cnt
    display cnt " bottles of beer on the wall"
    add 1 to cnt
    display space
  end-perform.
```

Forth

https://en.wikipedia.org/wiki/Forth_(programming_language)

Forth is a stack based language, dating from 1970.

It has very little syntax. It allows defining dictionaries of "words" which map to functions or data structures. The functions then operate on the stack.

There are still multiple implementations of Forth in use, and it can be very useful in environments with few resources (for instance, not much memory).

I have never programmed Forth myself, but I have (a long time ago) written PostScript, which has some similarities.

From the slides:

A very simple Forth example from https://www.whoishostingthis.com/resources/forth-programming/

```
: OUTMATH             Output a mathematical calculation
  ." We will now calculate: (2 + 3) * 5" CR
  2 3 + 5 *
  ." This equals: " . CR ;

OUTMATH
```

```
We will now calculate: (2 + 3) * 5
This equals: 25
```

In that example:

- `: OUTMATH` starts the definition of the word `OUTMATH`
- `.` <string> `CR`
    - `CR` outputs a new line ("carriage return")
- `2 3 + 5 *`
    1. puts 2 on the stack
    2. puts 3 on the stack
    3. takes the top two elements off the stack (2 and 3), adds them and puts the result (5) back on the stack
    4. puts 5 on the stack
    5. takes the top two elements off the stack (5 and 5), multiplies them and puts the result back on the stack
- `.` <string> `.` `CR` `;`
    - ...
- `OUTMATH` executes the newly define word

99 bottles in Forth from https://rosettacode.org/wiki/99_Bottles_of_Beer#Forth

```
:noname   dup . ." bottles" ;
:noname       ." 1 bottle"  ;
:noname ." no more bottles" ;
create bottles , , ,

: .bottles  dup 2 min cells bottles + @ execute ;
: .beer     .bottles ."  of beer" ;
: .wall     .beer ."  on the wall" ;
: .take     ." Take one down, pass it around" ;
: .verse    .wall cr .beer cr
        1- .take cr .wall cr ;
: verses    begin cr .verse ?dup 0= until ;

99 verses
```

or create a beer language amd use it, also from https://rosettacode.org/wiki/99_Bottles_of_Beer#Forth

```
DECIMAL
: BOTTLES ( n -- )
```

```
        DUP
        CASE
        1 OF    ." One more bottle " DROP ENDOF
        0 OF    ." NO MORE bottles " DROP ENDOF
                . ." bottles "    \ DEFAULT CASE
        ENDCASE ;

: ,    [CHAR] , EMIT  SPACE 100 MS CR ;
: .    [CHAR] . EMIT  300 MS  CR CR CR ;
: OF        ." of "   ;     : BEER     ." beer " ;
: ON        ." on "   ;     : THE      ." the "  ;
: WALL      ." wall" ;      : TAKE     ." take " ;
: ONE       ." one "  ;     : DOWN     ." down, " ;
: PASS      ." pass " ;     : IT       ." it "   ;
: AROUND    ." around" ;
: POPONE     1 SWAP CR ;
: DRINK      POSTPONE DO ; IMMEDIATE
: ANOTHER    S" -1 +LOOP" EVALUATE ; IMMEDIATE
: HOWMANY    S" I " EVALUATE ; IMMEDIATE
: ONELESS    S" I 1- " EVALUATE ; IMMEDIATE
: HANGOVER     ." :-("  CR QUIT ;

: BEERS ( n -- )   \ Usage:  99 BEERS
      POPONE
      DRINK
        HOWMANY BOTTLES OF BEER ON THE WALL ,
        HOWMANY BOTTLES OF BEER ,
        TAKE ONE DOWN PASS IT AROUND ,
        ONELESS BOTTLES OF BEER ON THE WALL .
      ANOTHER
      HANGOVER ;
```

---

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/f.html#Forth

```
\ Forth version of the 99 Bottles program.
\ Dan Reish, dreish@izzy.net

: .bottles ( n -- n-1 )
  dup 1 = IF  ." One bottle of beer on the wall," CR
             ." One bottle of beer," CR
             ." Take it down,"
  ELSE  dup . ." bottles of beer on the wall," CR
        dup . ." bottles of beer," CR
        ." Take one down,"
  THEN
  CR
  ." Pass it around," CR
  1-
  ?dup IF  dup 1 = IF  ." One bottle of beer on the wall;"
           ELSE  dup . ." bottles of beer on the wall;"
           THEN
        ELSE  ." No more bottles of beer on the wall."
  THEN
  CR
```

```
;

: nbottles ( n -- )
  BEGIN  .bottles  ?dup NOT UNTIL
;

99 nbottles
```

https://rosettacode.org/wiki/99_Bottles_of_Beer#Forth

```
:noname   dup . ." bottles" ;
:noname       ." 1 bottle"  ;
:noname ." no more bottles" ;
create bottles , , ,

: .bottles  dup 2 min cells bottles + @ execute ;
: .beer     .bottles ."  of beer" ;
: .wall     .beer ."  on the wall" ;
: .take     ." Take one down, pass it around" ;
: .verse    .wall cr .beer cr
        1- .take cr .wall cr ;
: verses    begin cr .verse ?dup 0= until ;

99 verses
```

or create a beer language and write the program:

```
DECIMAL
: BOTTLES ( n -- )
        DUP
        CASE
        1 OF    ." One more bottle " DROP ENDOF
        0 OF    ." NO MORE bottles " DROP ENDOF
                . ." bottles "    \ DEFAULT CASE
        ENDCASE ;

: ,   [CHAR] , EMIT  SPACE 100 MS CR ;
: .   [CHAR] . EMIT  300 MS  CR CR CR ;

: OF        ." of "   ;     : BEER     ." beer " ;
: ON        ." on "   ;     : THE      ." the "  ;
: WALL      ." wall" ;      : TAKE     ." take " ;
: ONE       ." one "  ;     : DOWN     ." down, " ;
: PASS      ." pass " ;     : IT       ." it "   ;
: AROUND    ." around" ;

: POPONE    1 SWAP CR ;
: DRINK     POSTPONE DO ; IMMEDIATE
: ANOTHER   S" -1 +LOOP" EVALUATE ; IMMEDIATE
: HOWMANY   S" I " EVALUATE ; IMMEDIATE
: ONELESS   S" I 1- " EVALUATE ; IMMEDIATE
: HANGOVER    ." :-(" CR QUIT ;

: BEERS ( n -- )   \ Usage:  99 BEERS
      POPONE
      DRINK
```

```
        HOWMANY BOTTLES OF BEER ON THE WALL ,
        HOWMANY BOTTLES OF BEER ,
        TAKE ONE DOWN PASS IT AROUND ,
        ONELESS BOTTLES OF BEER ON THE WALL .
      ANOTHER
      HANGOVER ;
```

The Algols - a selection

ALGOL 60 - Tony Hoare said "Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors."

ALGOL 68 - seen at the time as a very complex language

ALGOL W - Wirth's proposed successor to ALGOL 60, ancestor of PASCAL and Modula-2

Simula 67 - ALGOL 60 with classes

Ada - designed for safety and developing large systems

Stropping

https://en.wikipedia.org/wiki/Stropping_(syntax)

Nowadays, we're used to programming languages having reserved keywords. For instance, in Python you can't have a variable called `def` or `for`.

But as we've already seen with FORTRAN IV, that need not be the case - FORTRAN decided its keywords based on context.

In the ALGOL derived languages, it was common to use **stropping** to delimit keywords.

In the ALGOL languages, bold text would be used for keywords in documentation:

> **int** a real int = 3;

At the time, that was impossible to use in actual program texts.

Stropping (from "apostrophe") uses extra characters to mark keywords.

ALGOL 60 used QUOTE stropping

```
'INT' intval = 3;
```

ALGOL 68 typically used UPPER stropping

```
INT a real int = 3;
```

If the character set was limited to 6 bits, then there was only one case, so POINT stropping could be used:

```
.INT A REAL INT = 3;
```

Algol 68 could also use RES "stropping"; reserved words, as we'd expect

```
int a_real_int = 3;  # there are 61 accepted reserved words #
```

And, as the wikipedia page explains, there were other approaches as well.

Algol 68: UPPER stropping

```
# Add an element to the end of the list #
PROC append = ( REF LIST list, ELEMENT val ) VOID:
BEGIN
  IF list IS empty
  THEN
    list := HEAP NODE := ( val, empty )
  ELSE
    REF LIST tail := list;
    WHILE next OF tail ISNT empty
    DO
      tail := next OF tail
    OD;
    next OF tail := HEAP NODE := ( val, empty )
  FI
END;
```

APL and J

IBM Selectric and golfball (picture would be nice) are mentioned on the APL wikipedia page.

https://en.wikipedia.org/wiki/APL_(programming_language)#Mathematical_notation

> A mathematical notation for manipulating arrays was developed by Kenneth E. Iverson, starting in 1957 at Harvard University. In 1960, he began work for IBM where he developed this notation with Adin Falkoff and published it in his book A Programming Language in 1962.

Early implementations had to use English reserved words for functions and operators.

https://en.wikipedia.org/wiki/APL_(programming_language)#Hardware

> A key development in the ability to use APL effectively, before the wide use of cathode ray tube (CRT) terminals, was the development of a special IBM Selectric typewriter interchangeable typing element with all the special APL characters on it. This was used on paper printing terminal workstations using the Selectric typewriter and typing element mechanism, such as the IBM 1050 and IBM 2741 terminal. Keycaps could be placed over the normal keys to show which APL characters would be entered and typed when that key was struck. For the first time, a programmer could type in and see proper APL characters as used in Iverson's notation and not be forced to use awkward English keyword representations of them. Falkoff and Iverson had the special APL Selectric typing elements, 987 and 988, designed in late 1964, although no APL computer system was available to use them. Iverson cited Falkoff as the inspiration for the idea of using an IBM Selectric typing element for the APL character set.
>
> Many APL symbols, even with the APL characters on the Selectric typing element, still had to be typed in by over-striking two extant element characters. An example is the grade up character, which had to be made from a delta (shift-H) and a Sheffer stroke (shift-M). This was necessary because the APL character set was much larger than the 88 characters allowed on the typing element, even when letters were restricted to upper-case (capitals).

APL -> J, using ASCII with digraphs instead of special symbols (basically, it adds dot and colon to things to make new symbols)

APL -> S, a stastical programming language

R is an implementation of S with some extensions. Much S code should run unaltered.

https://rosettacode.org/wiki/99_Bottles_of_Beer#APL

Classic version:

```
bob  ←  { (⍕⍵), ' bottle', (1=⍵)↓'s of beer'}
bobw ←  {(bob ⍵) , ' on the wall'}
beer ←  { (bobw ⍵) , ', ', (bob ⍵) , '; take one down and pass it around, ', bobw ⍵-1}
↑beer¨ ⌽(1-⎕IO)+⍳99
```

and its equivalent in J

https://rosettacode.org/wiki/99_Bottles_of_Beer#J

```
bob =: ": , ' bottle' , (1 = ]) }. 's of beer'"_
bobw=: bob , ' on the wall'"_
beer=: bobw , ', ' , bob , '; take one down and pass it around, ' , bobw@<:
beer"0 >:i.-99
```

- https://en.wikipedia.org/wiki/APL_(programming_language)
- https://en.wikipedia.org/wiki/J_(programming_language)

Initially designed as a language for thinking about problems, and described in the book "A Programming Language" in 1962. It was used as a notation for thinking about problems, such as describing computer systems.

The first use of an implementation using actual APL symbology was in 1966.

---

https://en.wikipedia.org/wiki/APL_(programming_language)#Mathematical_notation

> A mathematical notation for manipulating arrays was developed by Kenneth E. Iverson, starting in 1957 at Harvard University. In 1960, he began work for IBM where he developed this notation with Adin Falkoff and published it in his book A Programming Language in 1962.

Early implementations had to use English reserved words for functions and operators.

https://en.wikipedia.org/wiki/APL_(programming_language)#Hardware

> A key development in the ability to use APL effectively, before the wide use of cathode ray tube (CRT) terminals, was the development of a special IBM Selectric typewriter interchangeable typing element with all the special APL characters on it. This was used on paper printing terminal workstations using the Selectric typewriter and typing element mechanism, such as the IBM 1050 and IBM 2741 terminal. Keycaps could be placed over the normal keys to show which APL characters would be entered and typed when that key was struck. For the first time, a programmer could type in and see proper APL characters as used in Iverson's notation and not be forced to use awkward English keyword representations of them. Falkoff and Iverson had the special APL Selectric typing elements, 987 and 988, designed in late 1964, although no APL computer system was available to use them. Iverson cited Falkoff as the inspiration for the idea of using an IBM Selectric typing element for the APL character set.
>
> Many APL symbols, even with the APL characters on the Selectric typing element, still had to be typed in by over-striking two extant element characters. An example is the grade up character, which had to be made from a delta (shift-H) and a Sheffer stroke (shift-M). This was necessary because the APL character set was much larger than the 88 characters allowed on the typing element, even when letters were restricted to upper-case (capitals).

APL -> J, using ASCII with digraphs instead of special symbols (basically, it adds dot and colon to things to make new symbols)

SNOBOL4

https://en.wikipedia.org/wiki/SNOBOL

SNOBOL was developed between 1962 and 1967 (SNOBOL4).

Introduced patterns as a first class datatype.

All SNOBOL command lines are of the form:

```
<label> <subject> <pattern> = <object> : <transfer>
```

All parts are optional.

- The <subject> is matched against the <pattern>.
- If <object> is present, any matched portion of <subject> is replaced with <object>
- <transfer> is then an absolute or conditional branch (to a <label>.
- A conditional branch is dependent upon the success/failure of evaluating the <subject>, <object> and <pattern>, the pattern match or the final assignment (to the <subject>).

So, for instance:

```
          OUTPUT = "What is your name?"
          Username = INPUT
          Username "J"                                                :S(LOVE)
          Username "K"                                                :S(HATE)
MEH       OUTPUT = "Hi, " Username                                    :(END)
LOVE      OUTPUT = "How nice to meet you, " Username                  :(END)
HATE      OUTPUT = "Oh. It's you, " Username
END
```

---

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/s.html#Snobol

```
* 99 BOTTLES OF BEER IN SNOBOL (UNTESTED)
        BEER = 99
MOREBEER OUTPUT = BEER ' BOTTLES OF BEER ON THE WALL'
        OUTPUT = BEER ' BOTTLES OF BEER'
        OUTPUT = 'TAKE ONE DOWN, PASS IT AROUND'
        BEER = BEER - 1
        OUTPUT = BEER ' BOTTLES OF BEER ON THE WALL'
        GT(BEER,0)   : S(MOREBEER)
        OUTPUT = 'NO MORE BOTTLES OF BEER ON THE WALL'
        OUTPUT = 'NO MORE BOTTLES OF BEER'
        OUTPUT = 'GO TO THE STORE AND BUY SOME MORE'
        OUTPUT = '99 BOTTLES OF BEER'
END
```

BCPL

https://en.wikipedia.org/wiki/BCPL

BCPL was first implemented in 1967.

- Systems level language
- The book of the language includes all the source code for the compiler
- BCPL was the first "brace" programming language, although it historically used `$( .. $)`.
- `IF .. DO ..` and `TEST .. THEN .. ELSE ..`

- The only datatype is the `word` - size depends on the computer
- Semicolons separate commands, and a semicolon at the end of a line may be omitted. To make this work, infix expression operators (+, etc.) may never start a line.

  Or, in other words, a command carries on over multiple lines when it ends with a character (+ or `,`, for instance) that implies as much.
- Labels are values, and one can do arithmetic on them
- An ancestor of C (CPL begat BCPL which begat B which begat C)

https://www.bell-labs.com/usr/dmr/www/bcpl.html - Martin Richards's BCPL Reference Manual, 1967

https://www.cl.cam.ac.uk/~mr10/bcplman.pdf - the BCPL user guide from 2020. Note that the examples use `{ .. }`.

Also:

- `$( .. )$`
- `IF .. THEN` and `TEST .. THEN .. ELSE`
- a statement continues to the next line if it can't have ended (so, for instance, if the last character was the + of an arithmetic expression
- labels *are* values, and since everything is a word, you can do arithmetic on them.

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/b.html#BCPL

```
// BCPL version of 99 Bottles of Beer.
// hacked by Akira KIDA <SDI00379@niftyserve.or.jp>
GET "LIBHDR"
MANIFEST $(
    BOTTLES = 99
$)
LET START() BE $(
    LET BEERS(N, S) BE $(
        TEST N = 0 THEN WRITEF("No more bottles")
                  ELSE WRITEF("%N bottle%S", N, (N = 1) -> "", "s")
        WRITEF(" of beer%S", S)
    $)
    FOR I = BOTTLES TO 1 BY -1 DO $(
            BEERS(I, " on the wall, ")
            BEERS(I, ".*NTake one down, pass it around.*N")
            BEERS(I - 1, " on the wall.*N")
    $)
    FINISH
$)
```

Prolog

Full stop to end expressions/statements, not semicolon

(I've heard people say Erlang is inspired by Prolog in some sense?)

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/e.html#Erlang

Prolog

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/p.html#Prolog

```
% 99 bottles of beer.
% Remko Troncon <spike@kotnet.org>

bottles :-
    bottles(99).

bottles(1) :-
    write('1 bottle of beer on the wall, 1 bottle of beer,'), nl,
    write('Take one down, and pass it around,'), nl,
    write('Now they are alle gone.'), nl.
bottles(X) :-
    X > 1,
    write(X), write(' bottles of beer on the wall,'), nl,
    write(X), write(' bottles of beer,'), nl,
    write('Take one down and pass it around,'), nl,
    NX is X - 1,
    write(NX), write(' bottles of beer on the wall.'), nl, nl,
    bottles(NX).
```

Prolog - works with SWI Prolog

https://rosettacode.org/wiki/99_Bottles_of_Beer/Prolog

```
bottles(0):-!.
bottles(X):-
    writef('%t bottles of beer on the wall \n',[X]),
    writef('%t bottles of beer\n',[X]),
    write('Take one down, pass it around\n'),
    succ(XN,X),
    writef('%t bottles of beer on the wall \n\n',[XN]),
    bottles(XN).

:- bottles(99).
```

or, handling plurals:

```
line1(X):- line2(X),write(' on the wall').
line2(0):- write('no more bottles of beer').
line2(1):- write('1 bottle of beer').
line2(X):- writef('%t bottles of beer',[X]).
line3(1):- write('Take it down, pass it around').
line3(X):- write('Take one down, pass it around').
line4(X):- line1(X).

bottles(0):-!.
bottles(X):-
    succ(XN,X),
    line1(X),nl,
    line2(X),nl,
    line3(X),nl,
    line4(XN),nl,nl,
    !,
    bottles(XN).

:- bottles(99).
```

S and R

https://rosettacode.org/wiki/99_Bottles_of_Beer#R

Simple looping solution in R

```
# a naive function to sing for N bottles of beer...
song = function(bottles){
  for(i in bottles:1){ #for every integer bottles, bottles-1 ... 1
    cat(bottles," bottles of beer on the wall \n",bottles," bottles of beer \nTake one down, pass it around \n",
        bottles-1, " bottles of beer on the wall \n"," \n" ,sep="")       #join and print the text (\n means new line)
        bottles = bottles - 1 #take one down...
  }
}
song(99)#play the song by calling the function
```

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/s.html#S-Plus

S - is this the right S?

```
Using S-Plus code

for(i in 100:1){
            if(i>1){
                        cat(i,"bottles of beer on the wall,",i,"bottles of beer\n")
                        cat("Take one down, pass it around\n")
                        cat(i-1,"bottles of beer on the wall\n",fill=TRUE)
            }
            else{
                        cat(i,"bottle of beer on the wall,",i,"bottle of beer\n")
                        cat("Take one down and pass it around\n")
                        cat("No bottles of beer on the wall!!\n",fill=TRUE)
            }
}
```

R

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/r.html#R

```
# R version of 99 Bottles of beer (Bottles.r)
# See http://www.r-project.org/ for more informations
# Philipp Winterberg, http://www.winterbergs.de

for (b in 99:1){
  print(b)
  print(" bottle(s) of beer on the wall,")
  print(b)
  print(" bottle(s) of beer.")
  print("Take one down, pass it around,")
  print(b-1)
  print(" bottle(s) of beer on the wall.")
  print("")
}
```

- https://en.wikipedia.org/wiki/S_%28programming_language%29
- https://en.wikipedia.org/wiki/R_(programming_language)

People here are probably more familiar with R, which is an implementation of S

APL -> S, a stastical programming language

R is an implementation of S with some extensions. Much S code should run unaltered.

- https://en.wikipedia.org/wiki/R_(programming_language) - initial release 1995
- https://en.wikipedia.org/wiki/S_(programming_language) - first working version in 1976

  Richard Becker's A Brief History of S indicates that they were very well aware of APL, but clearly S is not a descendant of APL.

APL in R by Jan de Leeuw and Masanao Yajima, 2016, is an online book that presents R code for APL array operations.

Smalltalk

https://en.wikipedia.org/wiki/Smalltalk

Smalltalk-80 was made available in 1980.

- Almost no syntax
- Still alive (for instance, Pharo)
- Influences everywhere
- http://www.jera.com/techinfo/readingSmalltalk.pdf "Reading Smalltalk"

Almost no syntax

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/s.html#SmallTalk

```
"Programmer: patrick m. ryan - pryan@access.digex.net
"http://www.access.digex.net/~pryan

99 to: 1 by: -1 do: [ :i |
        i print. ' bottles of beer on the wall, ' print.
        i print. ' bottles of beer. ' print.
        'take one down, pass it around, ' print.
        (i-1) print. ' bottles of beer on the wall, ' print.
```

I think that's rather elegant.

https://rosettacode.org/wiki/99_Bottles_of_Beer#Smalltalk

A straightforward approach

```
Smalltalk at: #sr put: 0 ; at: #s put: 0 !
sr := Dictionary new.
sr at: 0 put: ' bottle' ;
  at: 1 put: ' bottles' ;
  at: 2 put: ' of beer' ;
  at: 3 put: ' on the wall' ;
  at: 4 put: 'Take one down, pass it around' !
99 to: 0 by: -1 do: [:v | v print.
        ( v == 1 ) ifTrue: [ s := 0. ]
                   ifFalse: [ s := 1. ].
        Transcript show: (sr at:s) ; show: (sr at:2) ; show: (sr at:3) ; cr.
                   v print.
        Transcript show: (sr at:s) ; show: (sr at:2) ; cr.
```

```
                     (v ~~ 0) ifTrue: [ Transcript show: (sr at:4) ; cr. ].
  ].
```

https://pharo.org/ - squeak variant

ABC

For old times take

This is the programming language that Guido van Rossum worked on before inventing Python, and his experiences with ABC were significant in how he designed Python.

http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/a.html#ABC

```
<a href=http://www.cwi.nl/cwi/projects/abc.html>ABC</a> was developed
at CWI in the Netherlands.
PUT "by Whitey (whitey@netcom.com) - 10/13/96" IN author

HOW TO RETURN verse n:
  SELECT:
      n = 0:
        PUT "no more bottles of beer" IN s
      n = 1:
        PUT "1 bottle of beer" IN s
      ELSE:
        PUT "`n` bottles of beer" IN s
  RETURN s

HOW TO DRINK:
  PUT 99 IN num
  WHILE num > 0:
      WRITE verse num, " on the wall, ", verse num, "," /
      WRITE "take one down, pass it around," /
      PUT num - 1 IN num
      WRITE verse num, " on the wall." /

DRINK
```

Python

Just to show the "99 bottles" solutions, to give an idea of how much / how little those really convey about a programming language.

One "traditional"

https://rosettacode.org/wiki/99_Bottles_of_Beer/Python

```python
def sing(b, end):
    print(b or 'No more','bottle'+('s' if b-1 else ''), end)

for i in range(99, 0, -1):
    sing(i, 'of beer on the wall,')
    sing(i, 'of beer,')
    print('Take one down, pass it around,')
    sing(i-1, 'of beer on the wall.\n')
```

(mainly included to show how one should not necessarily judge a language from the examples given!)

And another that just misses the whole point of the exercise, but is definitely my favourite:

http://rosettacode.org/wiki/99_Bottles_of_Beer#Python_3

```
"""Pythonic 99 beer song (maybe the simplest naive implementation in Python 3)."""

  REGULAR_VERSE = '''\
  {n} bottles of beer on the wall, {n} bottles of beer
  Take one down and pass it around, {n_minus_1} bottles of beer on the wall.

  '''

  ENDING_VERSES = '''\
  2 bottles of beer on the wall, 2 bottles of beer.
  Take one down and pass it around, 1 bottle of beer on the wall.

  1 bottle of beer on the wall, 1 bottle of beer.
  Take one down and pass it around, no more bottles of beer on the wall.

  No more bottles of beer on the wall, no more bottles of beer.
  Go to the store and buy some more, 99 bottles of beer on the wall.

  '''
  for n in range(99, 2, -1):
      print(REGULAR_VERSE.format(n=n, n_minus_1=n - 1))
  print(ENDING_VERSES)
```

History and Timelines

- https://www.scriptol.com/programming/history.php
- https://www.scriptol.com/programming/list-programming-languages.php
- https://www.scriptol.com/programming/sieve.php
- https://www.levenez.com/lang/

starts with Plankalkul ! but rather limited on the languages it lists

- https://media.timetoast.com/timelines/programming-languages-b4c706df-fef5-4b23-8d87-2b0a666150df
- http://rigaux.org/language-study/diagram.html - with some links to others

  Has 2 versions - a simplified one, and a more complete one
- http://www.digibarn.com/collections/posters/tongues/ComputerLanguagesChart.png from http://www.digibarn.com/collections/posters/tongues/ appears to be rather nice at first glance

Interesting links

Probably more for the notes than for the slides. Not necessarily entirely pertinent to this exact topic...

- https://www.hillelwayne.com/post/influential-dead-languages/ 10 Most(ly dead) Influential Programming Languages, 2020-03-25, Hillel Wayne
- https://www.vidarholen.net/~vidar/An_Empirical_Investigation_into_Programming_Language_Syntax.pdf An Empirical Investigation into Programming Language Syntax, Andreas Stefik and Susanna Siebert, 2013

  > Stefik, A. and Siebert, S. 2013. An empirical investigation into programming language syntax. *ACM Trans.Comput.Educ.* 13, 4, Article 19 (November 2013), 40 pages.

  I haven't read this yet

Other links

(may also be interesting)

- More on punched cards:
  - https://craftofcoding.wordpress.com/2017/01/28/read-your-own-punch-cards/ shows how to read the same punched card we use as an example
  - https://en.wikipedia.org/wiki/Punched_card
  - https://homepage.divms.uiowa.edu/~jones/cards/codes.html
  - "type" your own punch card: https://www.masswerk.at/keypunch/
- https://en.wikipedia.org/wiki/History_of_programming_languages
- https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(syntax) (perhaps too much information)
- http://www.99-bottles-of-beer.net doesn't seem to be working at the moment
- https://web.mit.edu/kenta/www/two/beer.html has Fortran IV, but the pages for each language are on `.net` and don't seem to work at the moment
- http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/ has Fortran IV and seems to work
- https://www.hillelwayne.com/equals-as-assignment/ Why Does "=" Mean Assignment? also by Hillel Wayne, from 2018
- FORTRAN IV
  - http://www.math-cs.gordon.edu/courses/cs323/FORTRAN/fortran.html
  - http://www.jaymoseley.com/hercules/fortran/fort_mini.htm
  - http://www.quadibloc.com/comp/fort03.htm some context with respect to FORTRAN II, and some talk on specifics of particular implenentations

  Still to look at:

  - https://hackaday.com/2015/10/26/this-is-not-your-fathers-fortran/1G

Don't forget the excellent http://www.softwarepreservation.org/ and particularly the http://www.softwarepreservation.org/projects page, which has links to many pages of programming language history, with a huge number of useful links.

- https://www.whoishostingthis.com/resources/apl/

99 Bottles examples

Taken from one of:

- https://rosettacode.org/wiki/99_Bottles_of_Beer/Lisp
- http://www.info.univ-angers.fr/pub/gh/hilapr/beers/schade/

Sum of squares from:

- https://rosettacode.org/wiki/Sum_of_squares

Full acknowledgements for each code source are in the notes.

Written in reStructuredText.

Converted to PDF using rst2pdf.

Source and associated slides at https://github.com/tibs/old-proglang-syntaxes-talk

1 I'll keep using upper-case to name the language, since historically that is how it was named, but note that modern Fortran is named using mixed-case.