Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1975

The Essential Design Criterion for Computer Languages: Software Science

M. H. Halstead

Report Number:
76-191

Halstead, M. H., "The Essential Design Criterion for Computer Languages: Software Science" (1975). *Department of Computer Science Technical Reports.* Paper 133.
https://docs.lib.purdue.edu/cstech/133

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

THE ESSENTIAL DESIGN CRITERION
FOR COMPUTER LANGUAGES: SOFTWARE SCIENCE

M. H. Halstead
Purdue University
CSD TR 191

Prepared as a Panelist's Position Paper for Dr. Herbert Maisel's Section at NCC-76.

Before the discovery of the science of thermodynamics, the conversion of energy to work was strictly an art. Only with the understanding provided by quantitative relationships did it become possible to know, for example, that the efficiency of a steam engine was limited, in an absolute sense, by the temperature difference between the condenser and the boiler.

A similar pre-scientific tradition has heretofor dominated the design of all computer languages. Our previous lack of quantitative relationships has by now resulted in the implementation of well over a hundred different languages, each intended to be an improvement over its predecessors.

With the understanding provided by the discovery of the natural science of software, the ad hoc tradition becomes obsolete; languages can be quantitatively evaluated, and improvements can now be measured in fundamental terms.

In the next ten minutes, I will show, in some detail, why this is so, and a bit of the evidence supporting this position. Accordingly, we must discuss two related properties, Program Level and Language Level.

For the first, I might simply state that it is now well established that the product of level, L, times volume, V, remains invariant when that program is translated from one language to another. Instead, however, I would rather show by example what is meant by that statement. We can start by taking that old standby, Euclid's Greatest Common Divisor algorithm, and analyzing various possible implementations of it. We will start with a simple procedure call in which the entire algorithm is implied by the function name and end with a table-look-up in which all

possible divisors have been pre-stored.

First we will obtain the counts of operators and operands, both unique and accumulated.

1. GCD as Procedure Call

GCD (A B DIV)

OPERATOR	FREQUENCY
1. GCD	1
2. ( )	1
$\eta_1 = 2$	$N_1 = 2$

OPERAND	FREQUENCY
1. A	1
2. B	1
3. DIV	1
$\eta_2 = 3$	$N_2 = 3$

2. GCD in Fortran

```
      IF (A.NE.0) GO TO 1
      GCD = B
      RETURN
1     IF (B.NE.0) GO TO 3
2     GCD = A
      RETURN
3     IG = A/B
      R = A - B * IG
      IF (R.EQ.0) GO TO 2
      A = B
      B = R
      GO TO 3
```

OPERANDS - FORTRAN

	OPERAND	FREQUENCY
1.	B	6
2.	A	5
3.	0	3
4.	R	3
5.	GCD	2
6.	IG	2
	$n_2 = 6$	$N_2 = 21$

OPERATORS - FORTRAN

	OPERATOR	FREQUENCY
1.	EOL	10
2.	=	6
3.	IF	3
4.	( )	3
5.	GO TO 3	2
6.	.NE.	2
7.	GO TO 1	1
8.	GO TO 2	1
9.	—	1
10.	*	1
11.	/	1
12.	.EQ.	1
	$\eta_1 = 12$	$N_1 = 32$

3. GCD - TABLE LOOK UP

(1000 rows by 1000 columns)

```
TABLE:  0,  0,  0,  0,  0,  ....
        0,  1,  1,  1,  1,  ....
        0,  1,  2,  1,  2,  ....
        0,  1,  1,  3,  1,  ....
        0,  1,  2,  1,  4   ....
        .   .   .   .   .   ....
        .   .   .   .   .   ....
```

GCD := TABLE [A, B]

TABLE LOOK UP

	OPERATOR	FREQUENCY
1	,	1 000 001
2	:	1
3	:=	1
4	[ ]	1
$\eta_1 = 4$		$N_1 = 1\ 000\ 004$

	OPERAND	FREQUENCY
1.	TABLE	1 000 001
2.	A	1
3.	B	1
4.	GCD	1
$\eta_2 = 4$		$N_2 = 1\ 000\ 004$

Summarizing the measured values from the preceeding three implementations, and adding the comparable values for implementations in PL/I, Algol and the assembly language of the CDC 6500, we have the following raw data.

GCD Parameter Counts

Language	η_1	η_2	N_2	N
Call	2	3	3	5
PL/I	9	6	21	52
Algol 58	10	6	21	52
Fortran	12	6	21	53
Assembler	14	9	41	86
Tab. Look-Up	4	4	1 000 004	2 000 008

From these raw data, we calculate the Program Level as:

$$L = \frac{2}{\eta_1} \frac{\eta_2}{N_2}$$

the Volume as:

$$V = N \log_2 \eta$$

and the Intellegence content as:

$$I = L \times V$$

with the following results.

GCD: Level, Volume, and Intellegence.

Language	L	V	I
Call	1.00	12	12
PL/I	.063	203	13
Algol 58	.057	208	12
Fortran	.048	221	11
Assembly	.031	389	12
Table L-U	.000 002	6 000 024	12

Here, as in the other cases which have been studied, the product of L times V doesn't change much from one language to another. It does, of course, rise more than linearly as the number of conceptually unique input/output parameters increases.

Now, as Gordon will show at a session this afternoon, the effort required to convert a non-procedural problem statement to a running program should, and apparently does, require a number of elementary mental discriminations given by E, where:

$$E = V/L$$

Since we can substitute I/L for V, the effort can also be expressed as:

$$E = I/L^2$$

by remembering that I is language independent, we see at one that the effort involved in programming a given problem varies as the inverse square of the program level.

For problems having the same value of I, this allows a quantitative comparison of two languages. It is a further property of L, however, that in addition to its variation from one language to another, it also varies inversely with I.

This effect can be avoided by considering the Language Level, λ, as:

$$\lambda = L \times I$$

where both the mean value and the variance of λ increase with the power of a language.

Using the parameter counts of Professor Zweben for computer languages, and of Professor Kulm for English prose, we have:

λ FOR SMALL SAMPLES

	Mean	Variance
English prose	2.16	.73
PL/I	1.53	.92
Algol 58	1.21	.74
Fortran	1.11	.83
Pilot	.92	.43
Assembly (CDC)	.88	.42

Since the product of L times V is I, and the product of L times I is λ, the effort can be expressed as:

$$E = I^3/\lambda^2$$

Again, since I represents only the inherent complexity of a problem independent of its implementation, we again have an inverse square law for Language Level.

While it is true that the higher the level, the less the effort for a fluent programmer, there is still a theoretical limitation. For each increase in level, there is an increase in the amount which a programmer must learn to develop fluency. Consequently, the optimum level for a procedural language must also depend upon the amount of programming to be done in it.

In summary, then, if efficiency of the human effort is the objective of a new language, either with respect to initial implementation, debugging, or maintenance, then that efficiency can be measured, and quantitatively compared with existing languages. All that is required is to write a set of small sample programs in the proposed language, count their values of η_1, η_2, N_1 and N_2, and from these calculate λ, or λ^2.

It is only prudent to note, however, that thermodynamics gave us only an understanding of the principles involved in steam engines, but not detailed design specifications. Similarly, software science gives us only the basic relationships, and not the design specifications for any language.