2

Technical Report

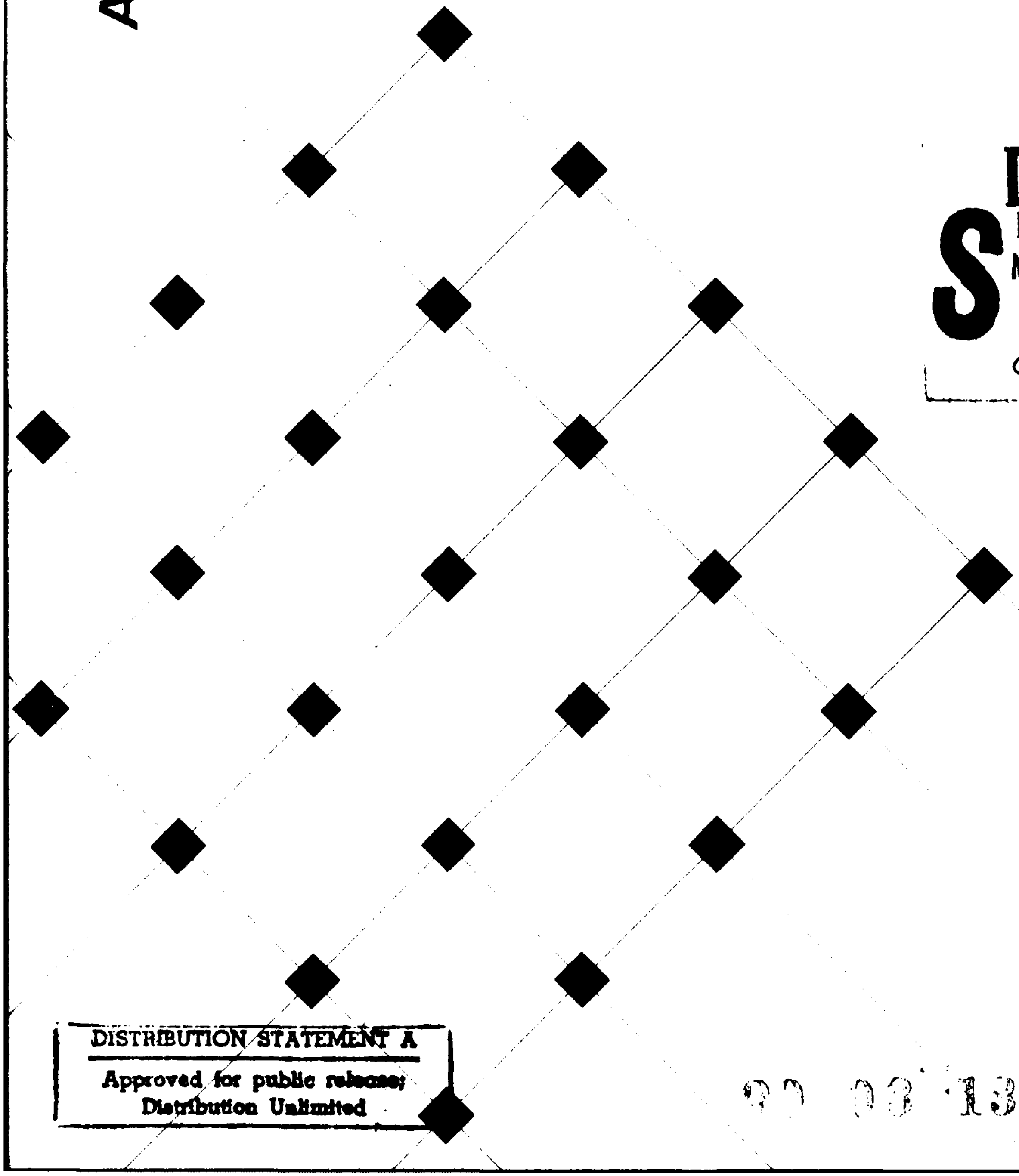
CMU/SEI-89-TR-30
ESD-TR-89-40

Carnegie-Mellon University
Software Engineering Institute

What a Software Engineer Needs to Know: I. Program Vocabulary

Mary Shaw
Dario Giuse
Raj Reddy

August 1989



Technical Report
CMU/SEI-89-TR-30
ESD-TR-89-40
August 1989

What a Software Engineer Needs to Know: I. Program Vocabulary

Mary Shaw
Software Engineering Institute
School of Computer Science
Carnegie Mellon University

Dario Giuse
Raj Reddy
School of Computer Science
Carnegie Mellon University

This report will also appear as Carnegie Mellon University School of Computer Science Technical Report No. CMU-CS-89-180.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Charles J Ryan

Charles J. Ryan, Major, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense, in part by Contract F19628-85-C0003 with the U.S. Air Force as the executive contracting agent, and in part by the Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4976, Amendment 20, under Contract F33615-87-C-1499, and ARPA Order No. 5167 under Contract N00039-85-C-0163, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Table of Contents

List of Figures

List of Tables

What a Software Engineer Needs to Know: I. Program Vocabulary

Abstract: Software development, like any complex task, requires a wide variety of knowledge and skills. We examine one particular kind of knowledge, the *programming language vocabulary* of the programmer, by gathering statistics on large bodies of code in three languages. This data shows that most of the identifiers in programs are either uses of built-in or standard library definitions or highly idiomatic uses of local variables. We interpret this result in light of general results on expertise and language acquisition. We conclude that tools to support the vocabulary component of software development are wanting, and this part of an engineer's education is at best haphazard, and we recommend ways to improve the situation.

1. Proficiency Requires Content Knowledge

Proficiency in any field requires a large store of facts together with a certain amount of context about their implications and appropriate use. The learning of these facts can be organized so that useful subsets are learned first, followed by more sophisticated subsets.

1.1. The Magic Number 70,000 ± 20,000

Experts know a great deal. This is true across a wide range of problem domains; studies demonstrate it for medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others [Reddy 88, pp. 13-14; Simon 89, p.1]. This is not of itself surprising. What is perhaps not so obvious is that the knowledge includes not only analytic techniques but also very large numbers of facts.

An often-quoted measure of factual knowledge is that an expert in any field must know 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived. Furthermore, in domains where there are full-time professionals, it takes no less than ten years for a world-class expert to achieve that level of proficiency [Simon 89 pp.2-4].

A software engineer's expertise includes facts about computer science in general, software design elements, programming idioms, representations, and specific knowledge about the program of current interest and about the language, environment, and tools in which this program is implemented. We are concerned here with the meanings of the symbols or identifiers that appear in the program and that name functions, variables, types, or other entities of the program.

1.2. The Content Component of Fluency

Proficiency requires content and context as well as skills. In the case of natural language fluency, for example, Hirsch argues that in American education abstract skills have driven out content. Students are expected to learn general skills from a few typical examples, not from the "piling up of information"; intellectual and social skills are supposed to develop naturally without regard to the specific content [Hirsch 88]. However, says Hirsch, specific information is important at all stages. Not only are the specific facts important in their own right, but they serve as carriers of shared culture and shared values. The accumulation of these shared symbols and their connotations supports the cooperation required for the complex undertakings of modern life.

Hirsch provides a list of some five thousand words and concepts that represent the information actually possessed by literate Americans. The list goes beyond simple vocabulary to enumerate objects, concepts, titles, and phrases that implicitly invoke cultural context beyond their dictionary definitions. Whether or not you agree in detail with its composition, the list and accompanying argument demonstrate the need for connotations as well as denotations of the vocabulary. Similarly, a programmer needs to know not only a programming language but also the system calls supported by the environment, the general-purpose libraries, the application-specific libraries, and how to combine invocations of these definitions effectively. Moreover, he or she must be familiar with the global definitions of the program of current interest and the rules about their use.

1.3. The Incremental Nature of Vocabulary

Natural language fluency is a particularly interesting case of proficiency. One indicator of fluency is the size of working vocabulary. Vocabulary development involves acquisition of both expanding sets of general vocabulary and of specialized vocabulary appropriate to particular domains. In the case of programming language proficiency, the general vocabulary includes the words of the programming language, the system calls of the environment, and various general-purpose subroutine libraries. The specialized vocabulary includes subroutine libraries specialized to an application domain and the definitions written specifically for a particular program.

1.3.1. General Vocabulary

English language fluency is acquired in stages [Curtis 87]:

1: "I never saw it before."

2: "I've heard of it, but I don't know what it means."

3: "I recognize it in context—it has something to do with ..."

4: "I know it."

The third stage provides a useful reading vocabulary—the ability to "get the gist" of a passage, but fourth-stage knowledge is required to write precisely.

English vocabulary is acquired both through vocabulary drill and through reading in context; by analogy, a programmer might study specifications or code of library routines, or alternatively might read large amounts of code that uses the routines of the library in order to see how they are used in algorithmic context. For English language vocabulary the rate of vocabulary growth cannot be accounted for by direct vocabulary instruction; it appears that much vocabulary is acquired by encountering words multiple times in context [Nagy 87].

Thorndike and Lorge [Thorndike 44] reported the frequency of occurrence of words in five large bodies of text. They found 1,069 words occurring at least 100 times per million words of text and another 952 occurring 50 to 99 times per million words of text. The number of words occurring at least once per million was 19,440; another 9,202 words occurred less than once per million but more often than four times per 18 million. Based on this list Thorndike and Lorge recommend for each grade level a number of words that students should learn as "a permanent part of their stock of word knowledge." Zipf, in his studies of languages, shows that in most natural languages the most common 100 words account for 50% of total usage and the most common 1,000 words account for about 85% of the usage [Zipf 49].

Because of very different interpretations of what it means to "know" a word. measures of vocabulary size have large variance. Taking these and other factors into account, however, Nagy and Herman [Nagy 87] estimate that a high school graduate can be expected to know around 40,000 words, acquiring them at a rate of around 3,000 words per year. This is consistent with Simon's observation that an expert takes ten years to acquire 50,000 chunks.

1.3.2. Technical Vocabulary

Johansson [Johansson 75] compares the word frequency of scientific English with that of other kinds of written English. Comparing the thousand most common words in a sample of 14,581 words of scientific Eng'ish with the thousand most common words in a sample of 50,406 words of general English, he found that about a third of the words were different. and that the differences occurred mainly in the second 500 words (ranked by frequency). It is not unreasonable to expect the phenomenon of a domain-specific vocabulary to appear in software engineering as well.

1.3.3. Growth of General and Technical Vocabulary

General vocabulary is acquired incrementally, largely during the school years. Imagine this acquisition as the mastery of increasing sets in a hierarchical, even graded, set of vocabulary lists. The specialized vocabulary of any particular field (be it hobby or profession) requires learning more new words and new definitions for old words. This specialized vocabulary component both overlaps the general vocabulary (i.e., words that are usually advanced become basic) and adds new words that are absent or rare in the general vocabulary.

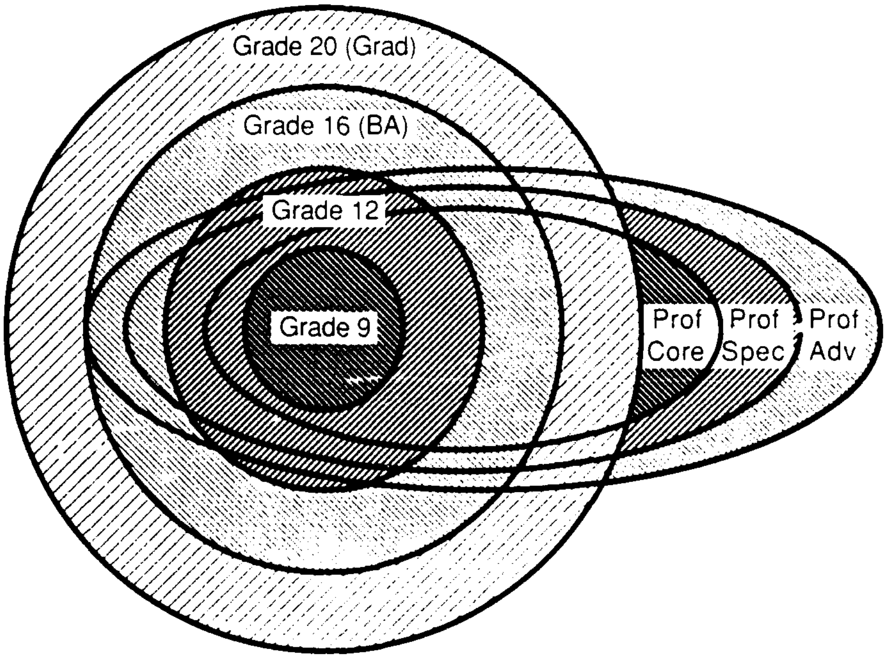


Figure 1: Vocabulary Families

1.4. Engineering Design Requires Content Knowledge

Engineering design problems differ in a number of significant ways; one of the most significant is the *distinction between routine and original design. Routine design involves* solving problems that resemble problems that have been solved before; it relies on reusing large portions of those prior solutions. Original design, on the other hand, involves finding creative ways to solve novel problems. The need for original design is much rarer than the need for routine design, so routine design is the bread and butter of engineering practice.

Most engineering disciplines capture, organize, and share design knowledge in order to make routine design simpler. Handbooks and manuals are often the carriers of this organized information [Marks 87, Perry 84].

Software development in most application domains tends to be more often original than routine—certainly more often original than would be necessary if we concentrated on capturing and organizing what is already known. One path to increased productivity is identifying applications that should be made routine and developing appropriate support. The current emphasis on reuse [Biggerstaff 89] emphasizes capturing and organizing existing knowledge. Indeed, subroutine libraries—especially libraries of operating system calls and general-purpose mathematical routines—have been a staple of programming for decades. But this knowledge cannot be useful if programmers don't know about it or aren't encouraged to use it, and library components require more care in design, implementation, and documentation than similar components that are simply embedded in systems.

The usual response to the problem of knowing what library definitions are available is improved indexing, classification, and search mechanisms. We suggest that these, like English dictionaries, are useful for recording little-used portions of the vocabulary and precise definitions of the core vocabulary. However, just as natural language fluency requires instant recognition of a core vocabulary, programming fluency should require an extensive vocabulary of definitions that the programmer can use familiarly, without regular recourse to documentation.

1.5. Software Development Requires Content Knowledge

Software must be understood by its creators and by its maintainers. Understanding software requires several qualitatively different kinds of knowledge:

- general knowledge about software
- general knowledge about the application domain
- the ability to use the language, operating system, methodology, and other software tools
- the requirements and motivation of the system
- the specific vocabulary of the particular software system, including knowledge and notation for:
    - the overall software architecture
    - interface protocols and interchange representations
    - algorithms and data structures
    - the code

This knowledge accounts for part of the 50,000 chunks of an expert's knowledge.

Most of these topics have been discussed at length, and many are included in a computer science curriculum. However, the ability of a programmer to read or write a program depends critically on his or her degree of mastery of the *vocabulary of the program*—that is, the meanings of the collection of variable names, reserved words, function names, and other lexical tokens that make up the program. This vocabulary includes constructs of several kinds:

1. built-in constructs of the language: reserved words, operators, syntactic connectors, etc.
2. names of standard library constructs, including functions and procedures, types, and data structures (including libraries for the application domain)
3. the shared vocabulary of this piece of software, for example as captured in the system dictionary
4. literals whose meaning is guaranteed to be given by the lexical token itself
5. local variables whose meaning is usually obvious from local context and is of no consequence in other parts of the program

Note that the vocabulary of the program is different from the programmer's vocabulary for programming, which includes many words that do not appear directly in a program.

The first three sorts of constructs should be part of the working vocabulary of any programmer who is to develop or modify the software. The third sort must be learned specifically for each system, but the first two are shared among many systems.

To see how significant the first two sorts of constructs are in the programming vocabulary, we examined a collection of programs to discover the distribution of "words" (i.e., lexical tokens) of each of these kinds. Our hypothesis is that linguistic analysis of program text can reveal the size and composition of the program vocabulary required of the developer or maintainer of a software system.

2. Empirical Observations: Vocabularies of 31 Programs

2.1. Approach

We examined 31 programs written in Ada, C, and Lisp. They ranged in size from 24 to 40,539 lines, with a mean of 5,947, and amounted in all to 184,351 lines. We counted the five kinds of constructs described in Section 1.5, counting them as built-in symbols, common library symbols, system-specific (user-defined) symbols with widespread use, literals, and purely local user-defined symbols using syntactic criteria to distinguish the classes. We counted both the number of distinct symbols and the number of uses of symbols. We did not distinguish among functions, subroutines, modules, types, and other sorts of identifiers except to the extent that such distinctions simplified the automatic data collection. Since we are interested in the program vocabularies rather than natural language, comments were stripped from the programs before making the counts.

2.2. Summary

Table 1 summarizes the data collected for these 31 programs and Figure 2 shows relations among vocabulary and usage of the five categories of symbols in the three languages. Overall, we found that built-in words and library names account for a small fraction of a program's discrete vocabulary but a very large share of the actual text. System-specific symbols and literals, however, constituted a somewhat larger fraction of the vocabulary than of actual use.

	Lisp		C		Ada	
Number of systems	8		11		12	
Total size (lines)	68,415		47,991		67,945	
Average size (lines)	8,552		4,363		5,662	
Range (lines)	317-40,539		926-11,240		24-19,648	
Average size (# symbols)	19,435		13,623		12,429	
Average vocabulary (# distinct symbols)	2,245		820		715	
Average usage rate (size/vocabulary)	8.7		16.5		17.4	
	N	%	N	%	N	%
Distinct instances (vocabulary size)						
Built-in symbols	469	3.0%	64	0.9%	115	1.9%
Common library symbols	498	3.2%	256	3.6%	28	0.5%
System-specific symbols	2,693	17.4%	1,414	19.6%	2,394	40.1%
Literals	7,394	47.7%	2,640	36.7%	1,030	17.3%
Purely local symbols	4,434	28.6%	2,824	39.2%	2,404	40.3%
Occurrences in the program						
Built-in symbols	46,208	29.7%	53,546	35.7%	62,559	41.9%
Common library symbols	7,227	4.6%	6,429	4.3%	4,492	3.0%
System-specific symbols	23,342	15.0%	10,785	7.2%	34,353	23.0%
Literals	28,106	18.1%	20,778	13.9%	11,600	7.8%
Purely local symbols	50,598	32.5%	58,311	38.9%	36,147	24.2%

Table 1: Summary Statistics

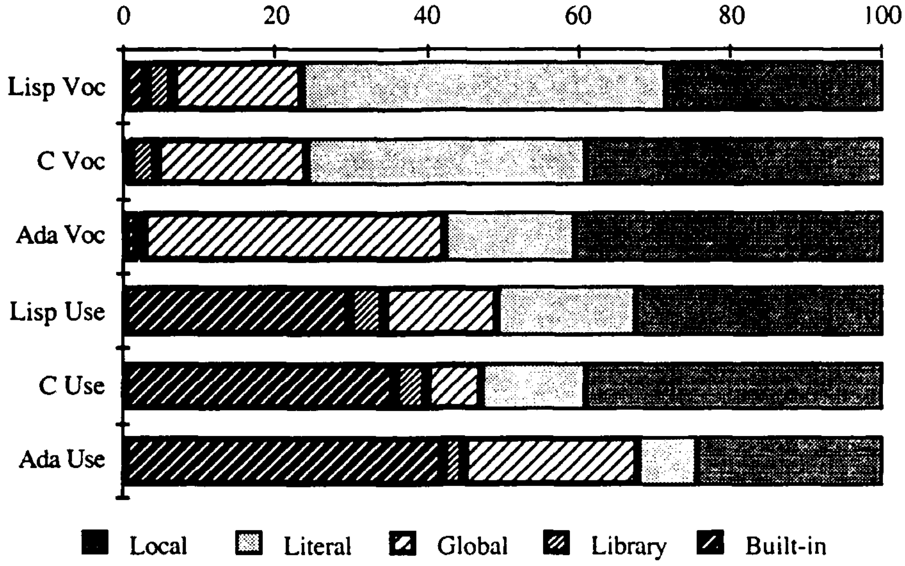


Figure 2: Overall Distribution of Symbols in Vocabulary (rows 1-3) and Usage (rows 4-6)

In summarizing the statistics from Tables 3 through 5, which present individual programs, the vocabulary size (that is, the count of instances) for each language was determined by taking unions of the vocabularies of the individual data cases. Since duplicates are counted when occurrences are tallied, the summaries for this portion of the data are the sums of the counts in the individual data cases.

The symbols in programs are of three kinds:

- *Meaning is lexically obvious:* The literals include numbers, quoted strings, and other symbols whose meaning (i.e., value) is given by their spelling. They account for 17% to 48% of the vocabulary but only 8% to 18% of the program text. In most cases essentially no effort is required to interpret them, though format strings in C and Lisp can be exceptions, as can the special syntax for literals that must be mapped to particular machine representations.
- *Meaning is syntactically obvious*: Purely local symbols have no meaning that persists over large scopes within the program. They account for 29% to 40% of the vocabulary and 24% to 39% of the program text. They are more heavily represented in the text of the Lisp programs than in the vocabulary for those programs, but the opposite is true for Ada—local symbols make up a drastically larger portion of the vocabularies than of the program text. The same is true to a lesser extent for C. Initial examination of the use of these variables indicates that they are often used idiomatically, with the syntactic template of the idiom carrying the meaning. Typical uses include:
  - temporary location for storage of an intermediate result of a computation, especially one that is used several times
  - accumulation of sum, product, or count
  - loop control
  - auxiliary pointer used to traverse a data structure

- status flag (boolean)
- formal parameter

Thus these local symbols often have semantics rather like pronouns; that is, they take on meaning from context, the meaning is easily understood within that context, and there is no reason to understand them outside their context of use. We did not distinguish semantically different (independently declared) uses in this study.

- *Meaning cannot be derived from local context*: The meanings of symbols in the remaining three categories must be learned, looked up, or derived. Call this the *learnable vocabulary* of the program. In the programs studied, symbols of this vocabulary are represented about twice as heavily in use as they are in the complete vocabulary; in C and Lisp they account for about 24% of the vocabulary and 47 to 49% of the uses and in Ada they account for 42% of the vocabulary and 68% of the uses. The contribution of each of the categories to the learnable vocabulary is interesting. Table 2 compares the distribution of distinct instances (*Vocab*) to occurrences (*Occur*) of the symbols in the learnable vocabulary.

	Lisp		C		Ada	
	Vocab	*Occur*	*Vocab*	*Occur*	*Vocab*	*Occur*
Count	3,660	76,777	1,734	70,760	2,537	101,404
Within the learnable vocabulary:						
% Built-in	12.8%	60.2%	3.7%	75.%	4.5%	61.7%
%Common library	13.6%	9.4%	14.8%	9.1%	1.1%	4.4%
%System-specific	73.6%	30.4%	81.5%	15.2%	94.4%	33.9%

Table 2: Learnable Vocabulary

This follows the pattern of English, in which the core vocabulary (the thousand most common words) accounts for 85% of the total text.

Expectedly, the built-in symbols are used much more heavily than any other group. Less obviously, the common library symbols are more heavily used than the system-specific symbols. Although these symbols make up but a small fraction of the Ada vocabulary, their rate of usage approximates that of the built-in symbols of Lisp. The ratio of frequency in the text to frequency in the vocabulary (from Table 2) is given in Table 3.

	Lisp	C	Ada
Built-in	4.7	20.6	13.5
Common library	0.7	0.6	4.0
System-specific	0.4	0.2	0.4

Table 3: Uses per Symbol in Learnable Vocabulary

We also note the relatively low number of symbols relative to the number of lines. On average Lisp programs have 2.3 symbols/line, C programs have 3.1 symbols/line, and Ada programs have 2.2 symbols/line. This is not explained by comments, because comments were removed before collecting the data. It is partly explained by blank lines used for readability, by many lines consisting simply of a subroutine call with one parameter, and, for C and Ada, by grouping symbols such as begin and end. This result is consistent with Knuth's [Knuth 71] finding that lines of Fortran text were quite sparse and that expressions were quite simple.

2.3. Details for Lisp Programs

Table 4 presents the statistics for 8 Lisp programs amounting to some 68,000 lines of code. The distribution of symbols among categories is shown for the vocabulary in Figure 3 and for overall usage in Figure 4. For Lisp, we defined the constructs of interest as follows:

- *Built-in symbols* : any of the 827 functions, macros, variables, or types defined in Steele's Common Lisp manual [Steele 84]
- *Common library symbols*: symbols neither defined as part of Common Lisp nor provided by the program
- *System-specific symbols*:
  - functions and macros defined by the program
  - symbols declared in an export statement
- *Literals*: numbers, quoted strings, character constants, symbols preceded by a quotation mark, keywords, and sharp macros
- *Purely local symbols*: local variables, special variables, and parameters

These programs were obtained from projects in the Carnegie Mellon University School of Computer Science. They cover a variety of applications and system functions; some are clients of others. They are:

- *hemlock.lisp*: the Hemlock text editor
- *inter.lisp*: graphics interactor package; uses kr.lisp and opal.lisp
- *kr.lisp*: frame-based knowledge representation system
- *lapidary.lisp*: object-oriented graphics editor; uses inter.lisp, opal,lisp, and kr.lisp
- *opal.lisp*: object-based graphics system; uses kr.lisp
- *ops.lisp*: Common-Lisp implementation of OPS-5
- *profile.lisp*: Lisp performance profiling tools
- *psgraph.lisp*: generates PostScript diagrams of arbitrary graphs

Common Lisp is remarkable for its large body of built-in operators. These originated as personal libraries for older Lisps. As time passed, some consensus emerged on the most commonly used functions, and these became system libraries. At that time the Lisp community developed a cultural expectation that a programmer would learn a system library as part of learning the language. Common Lisp took the next step toward unification and standardization and incorporated several hundred functions directly in the language. Further, even though these functions are incorporated in the language, Lisp programs remain heavy users of external libraries.

The pattern of usage of functions and variables is rather different for Lisp than for C and Ada. Lisp emphasizes the use of functions rather than variables and much smaller local scopes. The joint effect of these two factors is to generate relatively larger numbers of functions which are known outside local scopes; in our statistics, this increases the usage rate of system-specific symbols.

	hemlock.lisp		inter.lisp		kr.lisp		lapidary.lisp	
Size (lines)	40,539		5,197		2,419		11,400	
Size (# lexical tokens)	89,851		13,063		3,928		23,959	
Vocab (# distinct tokens)	10,034		2,121		516		2,221	
	N	%	N	%	N	%	N	%
Distinct instances (vocabulary size)								
Built-in symbols	410	4.1%	135	6.4%	134	26.0%	151	6.8%
Common library symbols	307	3.1%	97	4.6%	0	0.0%	107	4.8%
System-specific symbols	1,502	15.0%	190	9.0%	101	19.6%	293	13.2%
Literals	5,188	51.7%	1,232	58.1%	120	23.3%	1,071	48.2%
Purely local symbols	2,627	26.2%	467	22.0%	161	31.2%	599	27.0%
Occurrences in the program								
Built-in symbols	27,524	30.6%	3,327	25.5%	1,566	39.9%	5,709	23.8%
Common library symbols	835	0.9%	1,299	9.9%	0	0.0%	4,132	17.2%
System-specific symbols	18,421	20.5%	828	6.3%	586	14.9%	977	4.1%
Literals	13,647	15.2%	3,646	27.9%	269	6.8%	7,301	30.5%
Purely local symbols	29,424	32.7%	3,963	30.3%	1,507	38.4%	5,840	24.4%

	opal.lisp		ops.lisp		profile.lisp		psgraph.lisp	
Size (lines)	3,594		4,457		317		492	
Size (# lexical tokens)	9,323		13,488		551		1,318	
Vocab (# distinct tokens)	1,087		1,525		190		263	
	N	%	N	%	N	%	N	%
Distinct instances (vocabulary size)								
Built-in symbols	154	14.2%	153	10.0%	79	41.6%	51	19.4%
Common library symbols	98	9.0%	0	0.0%	11	5.8%	0	0.0%
System-specific symbols	218	20.1%	382	25.0%	16	8.4%	6	2.3%
Literals	365	33.6%	266	17.4%	44	23.2%	131	49.8%
Purely local symbols	252	23.2%	724	47.5%	40	21.1%	75	28.5%
Occurrences in the program								
Built-in symbols	2,239	24.0%	5,028	37.3%	262	47.5%	553	42.0%
Common library symbols	946	10.1%	0	0.0%	15	2.7%	0	0.0%
System-specific symbols	870	9.3%	1,598	11.8%	45	8.2%	17	1.3%
Literals	2,167	23.2%	806	6.0%	83	15.1%	187	14.2%
Purely local symbols	3,101	33.3%	6,056	44.9%	146	26.5%	561	42.6%

Table 4: Statistics on Lisp programs

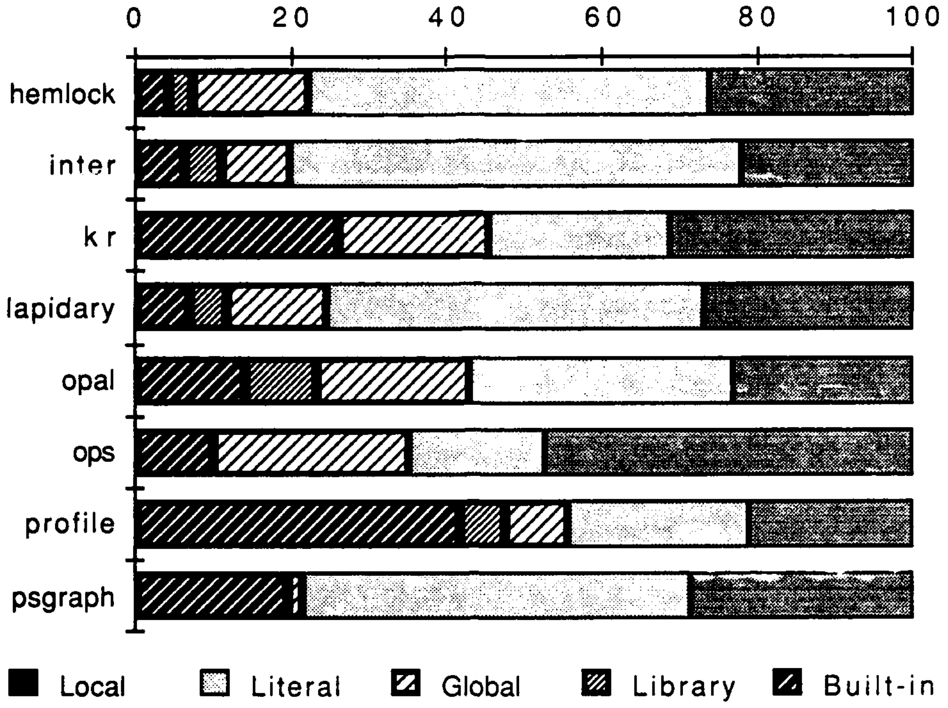


Figure 3: Lisp Vocabulary

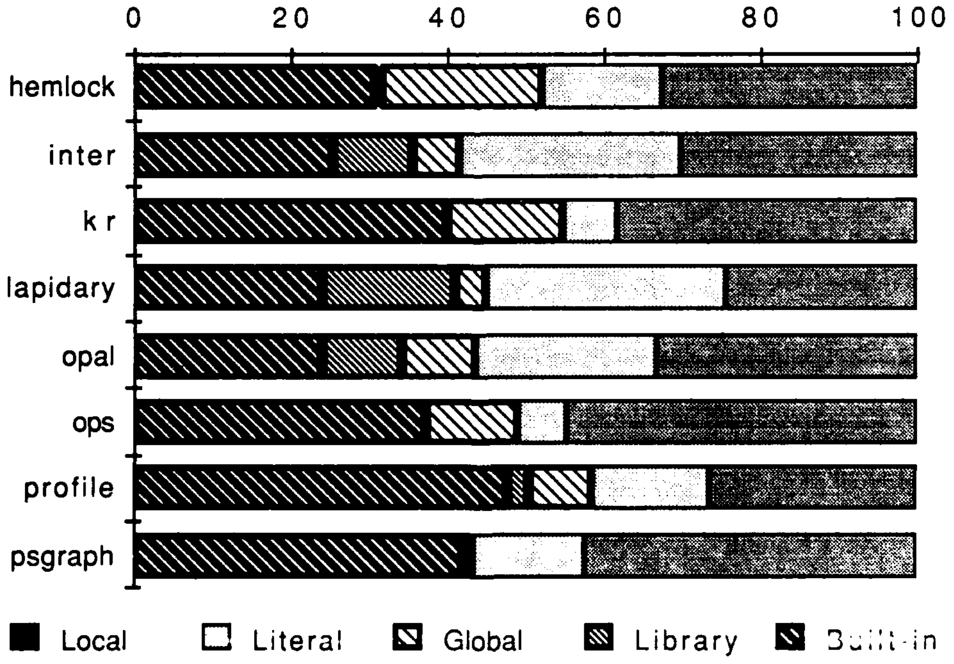


Figure 4: Lisp Usage

2.4. Details for C Programs

Table 5 presents the statistics for 11 C programs amounting to about 48,000 lines of code. The distribution of symbols among categories is shown for the vocabulary in Figure 5 and for overall usage in Figure 6. For C, we defined the constructs of interest as follows:

- *Built-in symbols* : the 64 operators and keywords of the C language
- *Common library symbols*: C library functions, macros, variables, or types as defined by all the .h files in the default directory for such definitions (/usr/include and /usr/include/sys); there are 3,743 such symbols on our system
- *System-specific symbols*:
    - user-defined functions, macros, variables, or types which are declared in .h files which are part of the program
    - any name explicitly mentioned in an extern statement
    - procedures or functions defined in the program
- *Literals*: numbers and quoted strings
- *Purely local symbols*: everything else

These programs, obtained from projects in the CMU School of Computer Science, cover a variety of applications and system functions. Except for the last four, they are largely self-contained and have relatively few high-level exports. This distinguishes them from the Lisp and Ada sets. They are:

- *avietest.c*: a collection of short test programs for testing the Mach operating system
- *chinese.c*: an interactive intelligent tutoring system for Chinese
- *dynload.c*: a dynamic loader for C programs
- *fscript.c*: a graphical window system
- *mololo.c*: a cartographical database and mapping system; it is monolithic and has no .h files
- *rfr.c*: a collection of very short performance test programs, standalone programs without .h files
- *sc.c*: a frame-based knowledge representation system
- *descartes.c*: a research prototype user interface construction system
- *airlog.c, ckbook.c, crostic.c*: example clients that use descartes.c

Despite C's relatively small built-in vocabulary, over 75% of the uses of words in the learnable vocabulary were of built-in words. The basic syntax of the language clearly contributes to this effect, but in comparison to Lisp it does not appear to be offset as much as one might expect by Lisp's large collection of built-in functions. Indeed, this may be the reason Lisp's occurrence rate of 60% is close to Ada's of 62%.

The programming culture of C is very much tied up with unix shell programming as well. We did not examine shell scripts; that would make an interesting extension of the study.

	avietest.c		chinese.c		dynload.c		fscript.c	
Size (lines)	4,187		5,248		1,328		9,867	
Size (# lexical tokens)	14,401		12,249		357		43,590	
Vocab (# distinct tokens)	513		1,395		425		1,930	
	N	%	N	%	N	%	N	%
Distinct Instances (vocabulary size)								
Built-in symbols	46	9.0%	57	4.1%	59	13.9%	61	3.2%
Common library symbols	45	8.8%	35	2.5%	43	10.1%	78	4.0%
System-specific symbols	58	11.3%	276	19.8%	50	11.8%	296	15.3%
Literals	167	32.6%	657	47.1%	105	24.7%	534	27.7%
Purely local symbols	197	38.4%	370	26.5%	168	39.5%	961	49.8%
Occurrences in the program								
Built-in symbols	5,764	40.0%	3,166	25.8%	1,548	41.2%	16,402	37.6%
Common library symbols	1,413	9.8%	419	3.4%	292	7.8%	688	1.6%
System-specific symbols	299	2.1%	2,659	21.7%	185	4.9%	2,081	4.8%
Literals	687	4.8%	1,629	13.3%	247	6.6%	9,658	22.2%
Purely local symbols	6,238	43.3%	4,376	35.7%	1,485	39.5%	14,761	33.9%

	mololo.c		rfr.c		sc.c		descartes.c	
Size (lines)	11,240		1,199		2,598		7,928	
Size (# lexical tokens)	40,082		4,103		5,583		15,538	
Vocab (# distinct tokens)	1,376		341		516		1,377	
	N	%	N	%	N	%	N	%
Distinct Instances (vocabulary size)								
Built-in symbols	45	3.3%	48	14.1%	46	8.9%	53	3.8%
Common library symbols	24	1.7%	31	9.1%	15	2.9%	50	3.6%
System-specific symbols	173	12.6%	19	5.6%	151	29.3%	364	26.4%
Literals	492	35.8%	88	25.8%	105	20.3%	481	34.9%
Purely local symbols	642	46.7%	155	45.5%	199	38.6%	429	31.2%
Occurrences in the program								
Built-in symbols	15,304	38.2%	1,764	43.0%	1,788	32.0%	4,692	30.2%
Common library symbols	1,459	3.6%	214	5.2%	145	2.6%	337	2.2%
System-specific symbols	664	1.7%	57	1.4%	1,067	19.1%	2,758	17.8%
Literals	5,410	13.5%	602	14.7%	390	7.0%	1,021	6.6%
Purely local symbols	17,245	43.0%	1,466	35.7%	2,193	39.3%	6,730	43.3%

	airlog.c		ckbook.c		crostic.c	
Size (lines)	2,344		926		1,126	
Size (# lexical tokens)	5,278		2,574		2,694	
Vocab (# distinct tokens)	563		222		403	
	N	%	N	%	N	%
Distinct Instances (vocabulary size)						
Built-in symbols	35	6.2%	31	14.0%	39	9.7%
Common library symbols	68	12.1%	31	14.0%	64	15.9%
System-specific symbols	102	18.1%	25	11.3%	45	11.2%
Literals	220	39.1%	46	20.7%	132	32.8%
Purely local symbols	138	24.5%	89	40.1%	123	30.5%
Occurrences in the program						
Built-in symbols	1,571	29.8%	761	29.6%	786	29.2%
Common library symbols	632	12.0%	401	15.6%	429	15.9%
System-specific symbols	622	11.8%	141	5.5%	252	9.4%
Literals	581	11.0%	284	11.0%	269	10.0%
Purely local symbols	1,872	35.5%	987	38.3%	958	35.6%

Table 5: Statistics on C Programs

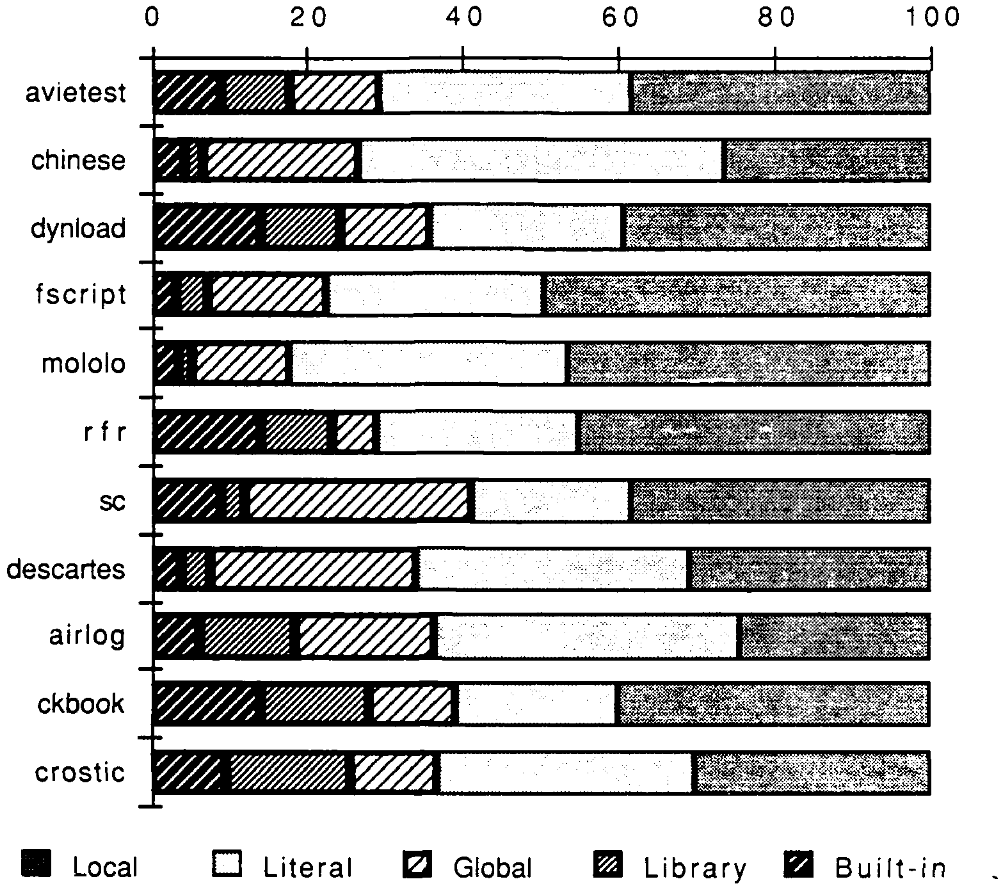


Figure 5: C Vocabulary

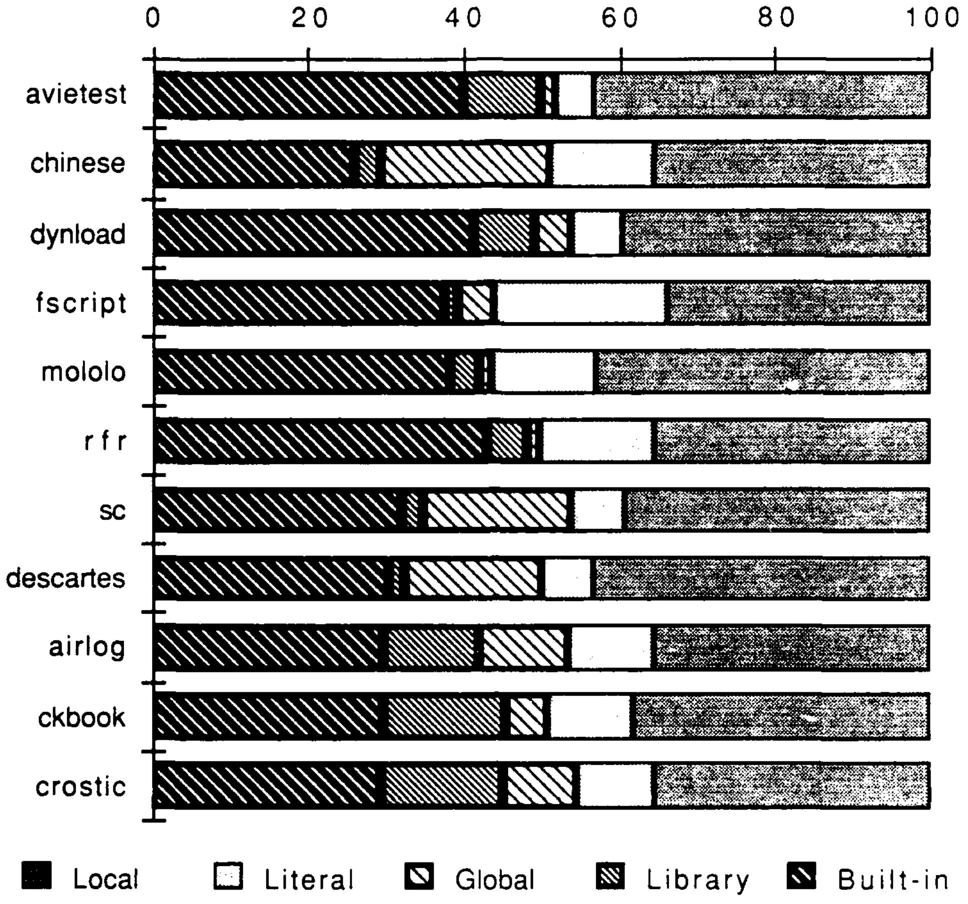


Figure 6: C Usage

2.5. Details for Ada Programs

Table 6 presents the statistics for 12 Ada programs amounting to nearly 68,000 lines of code. The distribution of symbols among categories is shown for the vocabulary in Figure 7 and for overall usage in Figure 8. For Ada, we defined the constructs of interest as follows:

- *Built-in symbols* : the 147 reserved words, attributes, operators, predefined types, exceptions, and pragmas of the Ada language
- *Common library symbols*: any symbol defined in one of the following packages: Text_IO, System, Calendar, string_pkg, string_scanner, and string_lists
- *System-specific symbols*: symbols defined in the specification part of a package, except for formal parameters to subroutines
- *Literals*: numbers and quoted strings
- *Purely local symbols*: other symbols, primarily local variables, and a few user types

The programs, obtained from the Ada Repository [Conn 87], cover a variety of applications and system functions. They are:

- *abstractions.ada*: a collection of abstract data structures (binary trees, etc)
- *ada-sql.ada*: files associated with the Standard Ada DBMS Interface (Ada/SQL)
- *alsptypes.ada*: an AI data types package (list processing and the like)
- *expert.ada*: a backward-chaining expert system
- *adafair85.ada*: a collection of testbench programs for the Ada Fair 1985
- *benchada.ada*: a collection of performance benchmarks
- *benchdhry.ada*: an Ada version of the Dhrystone testbench
- *benhtools.ada*: a short compilation testbench
- *benmath.ada*: a tiny mathematical benchmark program
- *bgt.ada*: a collection of performance/compilation benchmarks
- *piwga51.ada*: timing tools for performance measurement of the benchmarks
- *piwga831.ada*: a different version of the timing tools

The most striking difference between the Ada programs and the C programs is Ada's much lower usage rate for purely local symbols: 24% as against C's 39%. This is coupled to a higher usage rate for system-specific symbols: 23% for Ada but only 7% for C. The cause of the difference appears to be the extensive structuring of Ada programs via explicit package specifications. This is accompanied by a distribution of learnable vocabulary in which just under 5% of the symbols (the built-in vocabulary) account for 62% of the text (much like Lisp) and 94% of the symbols (the system-specific vocabulary) account for 34% of the text (a comparable rate of use per symbol as Lisp). This raises the following questions:

- What support should be made available for programmers to learn and understand the increased vocabulary of system-specific symbols?
- Would C programs that have been structured according to the discipline supported by Ada have statistics more like the C set or the Ada set?
- Should all the names in the Ada specifications really be exported?
- If this indicates a trend for future software, what tools will be required to help programmers cope with the system-specific vocabulary?

The Ada programs used substantially fewer library entries than did the Lisp and C programs. Two possible reasons come to mind:

- Ada is a relatively young language, and the community has not yet had time to reach consensus on what the shared libraries should be and to develop those libraries.
- The programs were obtained from a public repository, and the authors may have felt inhibited from assuming very much about the environment in which the code might subsequently run. This effect is, of course, magnified by the former.

The current emphasis on software reuse is especially strong in the Ada community. It will be interesting to see how any development of a component market affects the effective vocabulary of working programmers.

Because of the difference in application domains, it would be more likely for system data dictionaries to be available for the Ada programs than for the other programs. If data dictionaries had been available, it would have been useful to compare them to the list of symbols classified as system-specific vocabulary. That would be a useful extension of this study.

	abstractions.ada		ada-sql.ada		alsptypes.ada		expert.ada	
Size (lines)	1,781		19,648		6,278		1,048	
Size (# lexical tokens)	25,883		51,626		12,203		2,127	
Vocab (# distinct tokens)	1,484		2,066		603		22	
	N	%	N	%	N	%	N	%
Distinct Instances (vocabulary size)								
Built-in symbols	85	5.7%	87	4.2%	65	10.8%	63	28.0%
Common library symbols	20	1.3%	21	1.0%	10	1.7%	9	4.0%
System-specific symbols	658	44.3%	842	40.8%	140	23.2%	29	12.9%
Literals	227	15.3%	109	5.3%	130	21.6%	37	16.4%
Purely local symbols	494	33.3%	1,007	48.7%	258	42.8%	87	38.7%
Occurrences in the program								
Built-in symbols	11,941	46.1%	21,544	41.7%	4,161	34.1%	887	41.7%
Common library symbols	247	1.0%	713	1.4%	781	6.4%	78	3.7%
System-specific symbols	7,117	27.5%	13,349	25.9%	3,342	27.4%	270	12.7%
Literals	1,050	4.1%	3,546	6.9%	408	3.3%	154	7.2%
Purely local symbols	5,528	21.4%	12,474	24.2%	3.511	28.8%	738	34.7%

	adafair85.ada		benchada.ada		benchdhry.ada		benhtools.ada	
Size (lines)	8,553		2,330		571		353	
Size (# lexical tokens)	18,251		6,534		1,224		745	
Vocab (# distinct tokens)	1,167		405		201		190	
	N	%	N	%	N	%	N	%
Distinct Instances (vocabulary size)								
Built-in symbols	103	8.8%	67	16.5%	53	26.4%	49	25.8%
Common library symbols	14	1.2%	8	2.0%	5	2.5%	7	3.7%
System-specific symbols	560	48.0%	57	14.1%	36	17.9%	5	2.6%
Literals	374	32.0%	106	26.2%	38	18.9%	54	28.4%
Purely local symbols	116	9.9%	167	41.2%	69	34.3%	75	39.5%
Occurrences in the program								
Built-in symbols	7,690	42.1%	2,406	36.8%	472	38.6%	242	32.5%
Common library symbols	1,459	8.0%	371	5.7%	60	4.9%	67	9.0%
System-specific symbols	6,868	37.6%	398	6.1%	213	17.4%	19	2.6%
Literals	1,622	8.9%	1,047	16.0%	120	9.8%	82	11.0%
Purely local symbols	612	3.4%	2,312	35.4%	359	29.3%	335	45.0%

	benmath.ada		bgt.ada		piwga51.ada		piwga831.ada	
Size (lines)	24		2,436		4,701		4,189	
Size (# lexical tokens)	137		6,601		12,502		11,318	
Vocab (# distinct tokens)	19		436		918		862	
	N	%	N	%	N	%	N	%
Distinct Instances (vocabulary size)								
Built-in symbols	9	47.4%	63	14.4%	80	8.7%	80	9.3%
Common library symbols	0	0.0%	8	1.8%	12	1.3%	12	1.4%
System-specific symbols	7	36.8%	87	20.0%	145	15.8%	140	16.2%
Literals	2	10.5%	155	35.6%	198	21.6%	181	21.0%
Purely local symbols	1	5.3%	123	28.2%	483	52.6%	449	52.1%
Occurrences in the program								
Built-in symbols	91	66.4%	3,059	46.3%	5,260	42.1%	4,806	42.5%
Common library symbols	0	0.0%	206	3.1%	279	2.2%	231	2.0%
System-specific symbols	22	16.1%	843	12.8%	1,037	8.3%	875	7.7%
Literals	12	8.8%	868	13.1%	1,407	11.3%	1,284	11.3%
Purely local symbols	12	8.8%	1,625	24.6%	4.519	36.1%	4,122	36.4%

Table 6: Statistics on Ada Programs

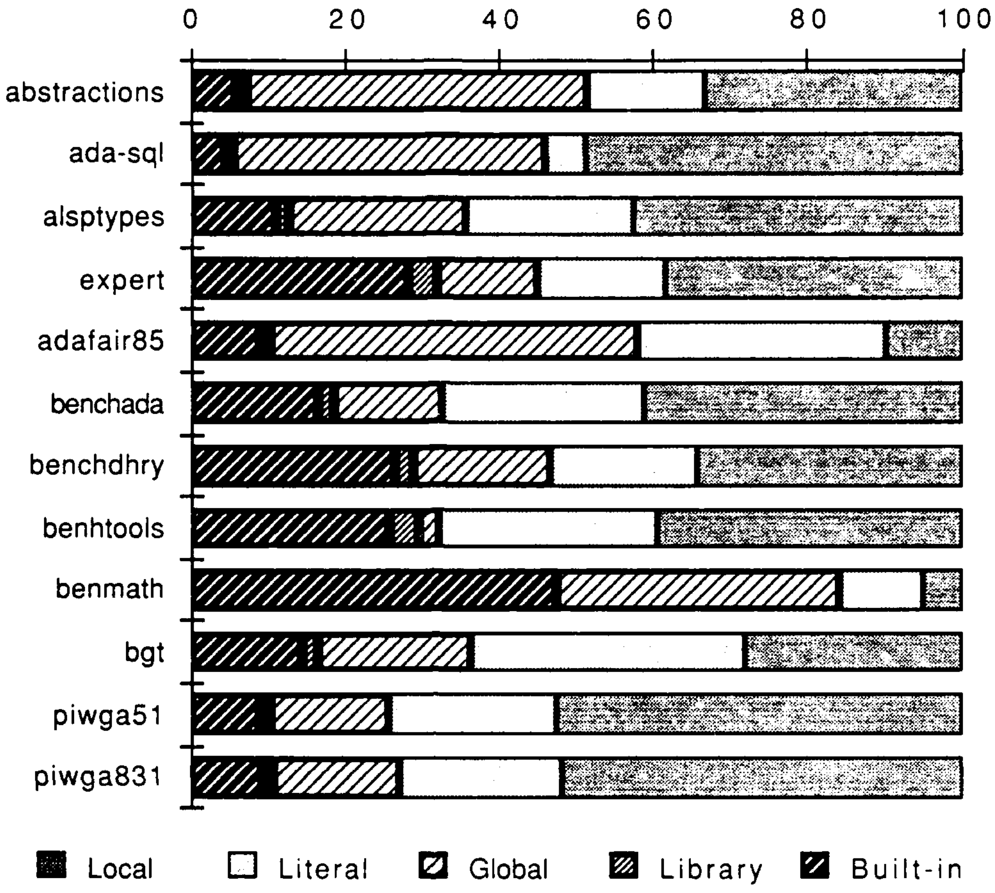


Figure 7: Ada Vocabulary

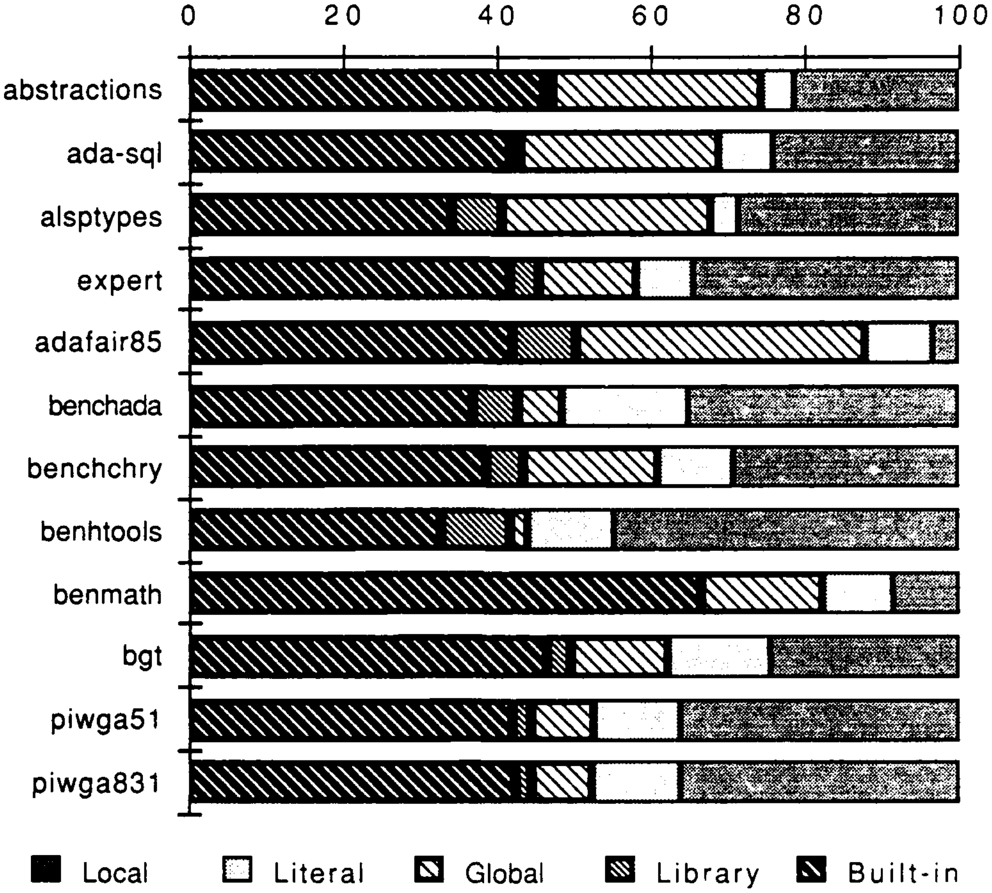


Figure 8: Ada Usage

3. Implications for Education and Practice

Both general studies of expertise and our data on particular programs indicate that a large body of facts is important to a working professional. We turn now to the question of how software engineers should acquire this knowledge, first as students and subsequently as working professionals.

Generally speaking, there are three ways to obtain a piece of information you need: you can remember it, you can look it up, or you can derive it. Each of these incurs costs of three kinds (over and above the cost of developing the knowledge itself): general overhead to the profession for creating the infrastructure that organizes the knowledge, initial cost for each professional to learn hᴖw to use the knowledge, and the direct cost each time the knowledge is used in practice. These costs have different distributions:

	Infrastructure Cost	*Initial Learning Cost*	*Cost of Use in Practice*
Memory	low	high	low
Reference	high	low	medium
Derivation	medium-high	medium	high

Memorization requires a relatively large initial investment in learning the material, which is then available for instant use. *Reference materials* require a large investment by the profession for developing both the organization and the content; each individual student must then learn how to use the reference materials and take the time to do so as a working professional. *Deriving information* may involve ad hoc creation from scratch, it may involve instantiation of a formal model, or it may involve inferring meaning from other available information; to the extent that formal models are available their formulation requires a substantial initial investment. Students first learn the models, then apply them in practice; since each new application requires the model to be applied anew, the cost in use may be quite high.

Each professional's allocation of effort among these alternatives is driven by what he or she has already learned, by habits developed during that education, and by the reference materials available. At present, general-purpose reference material for software is scarce, though documentation for specific computer systems, programming languages, and applications may be quite extensive. Even when extensive documentation is available, however, it may be under-used because it is poorly indexed or because software developers have learned to prefer fresh derivation to use of existing solutions.

Our concern here is with program vocabulary—that is, with the symbols of a program and their meaning. Access to information about these symbols is primarily through memory and through reference materials, though inference of meaning plays a significant role for local variables. Accordingly, we examine ways to improve access to the programming vocabulary with each of these mechanisms: *memory* for vocabulary acquisition, *reference* materials and tools for access, and *derivation* of meaning from context. We also address the implications for programming *productivity.*

3.1. Memory for Vocabulary Acquisition

3.1.1 How Vocabulary is Learned

For natural language, there is a strong correlation between vocabulary knowledge and reading comprehension. Both are in turn correlated with general background knowledge of the subject matter of the text. This background knowledge is an organized, interrelated structure, not an unstructured collection of facts; knowing where a word fits into this structure is an essential part of understanding it. It follows that simple drill in vocabulary does not contribute much to reading comprehension. Vocabulary instruction that emphasizes how new words fit into prior knowledge is useful, as is vocabulary instruction that presents new words in the context of a story. Further, vocabulary instruction can be made more effective by requiring students to manipulate words in varied ways, to encounter new words frequently while they are being taught, and to look for uses of the new words outside the classroom. Wide reading leads to expanded vocabulary, which in turn leads to better reading comprehension [Nagy 87, Beck 87] .

3.1.2 Recommendations: Read a Good Program, Write a Good Program

Elementary software engineering education emphasizes problem solving with the use of a programming language, design of algorithms and data structures, and specification and analysis of programs. There is considerable emphasis on the synthesis of new programs, some emphasis on useful design elements, and very little emphasis on the libraries and system calls available to the programmer. It is rare indeed for students to be expected to read significant passages of code that they did not themselves write. The theme is reasoning and design technique, not specific detailed knowledge of many different programs.

Our data on real programs of significant size indicates that professional software developers need substantial familiarity with the vocabulary of identifiers that appear in programs. The Ada data, in particular, suggests that a very small built-in vocabulary accounts for a large fraction of the program text and indeed of the text whose meaning is not obvious from context. Natural language vocabulary studies suggest that software developers should, as students, receive systematic education in this vocabulary.

Specifically, we recommend:

- Reading well-written programs that illustrate the concepts being taught and the supporting libraries and system calls in applications that the students can understand. (This would probably benefit other aspects of the education as well.)
- Studying the meanings of the external symbols of these programs.
- Using the system reference manuals and library documentation as dictionaries to support learning the meanings of the symbols, but not as a substitute for actual retention of meanings for the important symbols.
- Reinforcing this reading by expecting the correct use of the libraries and system calls in programming assignments.

3.2. Reference Materials and Tools for Access

3.2.1 The Role of Reference Materials

Not all material can or should be memorized; the choice among memory, reference, and derivation depends on the amount of use a piece of information will receive as well as on the cost of developing and learning it. Thus there will always be a use for reference materials. As noted above, most engineering disciplines rely heavily on such materials, especially in the form of handbooks. These handbooks organize large bodies of information into a form that is genuinely accessible to the practicing engineer. This accessibility depends not only on the structure of the material in the handbook, but also on the incorporation of the conceptual structure in the early education of the engineer.

In the case of the program vocabulary, reference material must help support both reading—the recognition vocabulary—and writing—the generation vocabulary. (These correspond to the third and fourth stages of language acquisition described in Section 1.3.1.) Each needs its own support, and the support should be on-line.

3.2.2 Recognition Vocabulary Supports Maintenance

The simpler of these two problems is support for the reading vocabulary. When reading a program it is often enough to know the approximate meaning of a symbol; it is frequently not necessary to know special restrictions on use, error conditions, etc. Fortunately, this is the aspect of the vocabulary task most important for software maintainers. Even when tracking down a bug, it is necessary to read portions of the program to establish context before examining precise usage of the constructs that may be implicated in the bug.

Prototype tools already exist to help readers of natural language prose; these tools allow the reader to point at a word and ask for help. The response may be simple delivery of a dictionary definition; more helpfully, the response may depend on the context in which the word is used and thus provide an explanation appropriate to the content.

Similar tools have also been built for software, for example in Smalltalk [Goldberg 80] and Cedar [Teitelman 84]. They allow such operations as pointing at a built-in symbol to get standard "help" text or pointing at a user-defined symbol to see the definition of that symbol. It should not be difficult to fetch elements of a data dictionary in the same way. Note that scope rules and overloading introduce subtleties in the implementation of these tools.

The ability to look up the definition of a symbol quickly does not eliminate the need to know the meanings of a core vocabulary without recourse to tools. It can, however, help with the learning of the core vocabulary and support the use of symbols too rare to deserve a place in the core vocabulary.

3.2.3 Generation Vocabulary is Needed for Development

Writing is quite a bit harder than reading. First, the reader is presented with a word to recognize, whereas the writer must generate an appropriate word for the intended meaning. Second, the reader need only recognize a meaning in context, whereas the writer must select a word that not only carries the intended meaning but also satisfies restrictions on the use of the word and avoids possible ambiguities. This problem is even more difficult in software development where, as in mathematics, a symbol may have quite detailed or complex restrictions on use.

There is by now a quite large body of code stored in libraries or otherwise available for reuse. In practice, the reuse of this code is inhibited by the programmer's difficulty in discovering what is there (generation of vocabulary) and in understanding the code's restriction on use (test of applicability). There may, for example, be mismatches between the library representation and the program's, between the range of values supported by the library and that needed by the program, and so on. The indexing and organization tasks that are already on the agenda for software reuse will help establish a basis for tools to help with the generation vocabulary. In addition, development of libraries is not unlike the invention of vocabulary: it is difficult to anticipate what vocabulary will actually turn out to be useful.

3.2.4. Recommendations: Know Where (and How) to Look It Up

The data reported here shows that many programs have a substantial vocabulary of system-specific definitions that both developers and maintainers must know. Although the most common of these should be mastered by anyone who will work extensively with the system, reference tools could help with the remainder. Specifically, we recommend:

- Expanding the development of readers' assistant tools for accessing a system's data dictionary and for looking up definitions of user-defined symbols.
- Augmenting the definition-retrieval tools to explain syntax and usage.
- Continuing the work on indexing and retrieval for software libraries, with emphasis on techniques that will work for libraries with thousands or tens of thousands of entries.
- Teaching the conceptual structure of the discipline assumed by these tools as part of early computer science education.

3.3. Derivation of Meaning from Context

Unlike natural language vocabulary, where the meaning of a word can often be inferred from its spelling and a general knowledge of similar words of the language, programming language vocabulary is usually dominated by words with arbitrary spelling. This is neither necessary nor desirable in the long run. However, programmers should know two ways of inferring meaning from context.

First, naming conventions are often used to indicate the meaning of an identifier. Mnemonic naming is the most-cited example of this, but perhaps more useful are the systematic inflections of a root name to indicate common predicates and operations, such as the -P suffix in Lisp to indicate a predicate. Conventions about parameter order can also be helpful.

Second, 25-40% of identifiers in programs are purely local. They are used idiomatically as loop counters, as temporary variables, list-tracers, formal parameters, and so on. There is no reason to remember the meanings of these variables for any longer than it takes to read the page of program text on which they appear; they have the same intellectual standing as pronouns. Programmers should learn these patterns as an aid to recognizing the meanings of these variables from context.

3.3.1. Recommendations: When in Rome, Talk As the Romans Do

Contextual understanding depends on shared conventions between readers and writers. A first step is to make these explicit and teach them. Specifically, we recommend:

- Identifying common idioms for use of local variables and cataloging them.
- Extending the tools for reading programs to understand these idiomatic usages.

3.4. Implications for Productivity

Productivity of software developers is a matter of national concern. The lessons from studies of expertise indicate that practitioners with more chunks of factual knowledge are more proficient. We conjecture that there is a correlation between programming productivity and fluency in the basic vocabulary of the software, both through memorization and through facile use of reference tools.

3.4.1. Recommendations: Try It, You'll Like It

The proposition that explicit efforts to learn the vocabulary of a program pay off in productivity should be tested in both development and maintenance environments. Specifically we recommend three experiments with the following general character:

- As a group of maintainers takes over a piece of software, convert some of the initial training of part of the group to carefully designed exercises in acquiring the vocabulary of the software system. Then compare the effectiveness of these maintainers with the effectiveness of maintainers trained in the usual way.
- Provide a set of program explanation tools to a group of software developers, measuring their effectiveness before and after adoption of these tools.
- Compare the rate at which students learn to program with and without deliberate attempts to expand their programming vocabularies by reading good programs, studying libraries, and using existing libraries in programming exercises.

4. References

[Beck 87] Isabel L. Beck, Margaret G. McKeown, and Richard C. Omanson.
"The effects and uses of diverse vocabulary instructional techniques."
In [McKeown 87], pp. 147-163.

[Biggerstaff 89] Ted J. Biggerstaff and Alan J. Perlis.
Software Reusability. Two volumes.
ACM Press, 1989.

[Conn 87] Richard Conn.
The Ada Software Repository and the Defense Data Network.
Crane Duplicating Services, West Barnstaple, MA, 1987.

[Curtis 87] Mary E. Curtis.
"Vocabulary testing and vocabulary instruction."
In [McKeown 87], pp. 37-51.

[Goldberg 80] Adele Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, 1984.

[Hirsch 88] E. D. Hirsch, Jr.
Cultural Literacy: What Every American Needs to Know.
Vintage Books, 1988.

[Johansson 75] Stig Johansson.
Some Aspects of the Vocabulary of Learned and Scientific English.
Gothenburg Studies in English 42.
Acta Universitatis Gothoburgensis, 1975.

[Knuth 71] Donald E. Knuth.
"An empirical study of Fortran programs."
Software—Practice and Experience 1(2): 105-133, April-June 1971.

[Marks 87] Lionel S. Marks et al.
Marks' Standard Handbook for Mechanical Engineers.
McGraw-Hill, 1987.

[McKeown 87] Margaret G. McKeown and Mary E. Curtis.
The Nature of Vocabulary Acquisition.
Lawrence Erlbaum Associates, 1987.

[Nagy 87] William E. Nagy and Patricia A. Herman.
"Breadth and depth of vocabulary knowledge: implications for acquisition and instruction."
In [McKeown 87] pp. 19-36.

[Perry 84] Robert H. Perry et al.
Perry's Chemical Engineers' Handbook.
McGraw-Hill,1984.

[Pressley 87] Michael Pressley, Joel R. Levin, and Mark A. McDaniel.
"Remembering versus inferring what a word means: mnemonic and contextual approaches."
In [McKeown 87], pp. 107-128.

[Reddy 88] Raj Reddy.
"Foundations and Grand Challenges of Artificial Intelligence. "
AI Magazine, 9,(4): 9-21, (Winter 1988).
(1988 presidential address, American Association for Artificial Intelligence)

[Simon 89] Herbert A. Simon.
"Human Experts and Knowledge-Based Systems."
Talk given at IFIP WG 10.1 Workshop on Concepts and Characteristics of Knowledge-Based Systems, Mt Fuji Japan, November 9-12,1987.

[Steele 84] G. L. Steele.
Common LISP—The Language.
Digital Press, 1984.

[Teitelman 84] Warren Teitelman.
"A tour through Cedar."
IEEE Software, April 1984, pp. 44-73.

[Thorndike 44] Edward L. Thorndike and Irving Lorge.
The Teacher's Word Book of 30,000 Words.
Teachers College Press, Columbia University, 1944.

[Zipf 49] G. K. Zipf.
Human Behavior and the Principle of Least Effort.
Addison-Wesley, 1949.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
NONE

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-89-TR-30

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-TR-89-40

6a. NAME OF PERFORMING ORGANIZATION
SOFTWARE ENGINEERING INST.

6b. OFFICE SYMBOL *(If applicable)*
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

6c. ADDRESS *(City, State and ZIP Code)*
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213

7b. ADDRESS *(City, State and ZIP Code)*
ESD/XRS1
HANSCOM AIR FORCE BASE
HANSCOM, MA 01731

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

8b. OFFICE SYMBOL *(If applicable)*
ESD/XRS1

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962885C0003

8c. ADDRESS *(City, State and ZIP Code)*
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE *(Include Security Classification)*
WHAT A SOFTWARE ENGINEER NEEDS TO KNOW: I. PROGRAM VOCABULARY

12. PERSONAL AUTHOR(S)
Mary Shaw, Dario Giuse, Raj Reddy

13a. TYPE OF REPORT
FINAL

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Yr., Mo., Day)*
August 1989

15. PAGE COUNT
28

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
software engineering
programming language vocabulary
software engineering education

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Software development, like any complex task, requires a wide variety of knowledge and skills. We examine one particular kind of knowledge, the programming language vocabulary of the programmer, by gathering statistics on large bodies of code in three languages. This data shows that most of the identifiers in programs are either uses of built-in or standard library definitions or highly idiomatic uses of local variables. We interpret this result in light of general results on expertise and language acquisition. We conclude that tools to support the vocabulary component of software development are wanting, and this part of a engineer's education is at best haphazard, and we recommend ways to improve the situation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. □ DTIC USERS ☒

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED, UNLIMITED DISTRIBUTION

22a. NAME OF RESPONSIBLE INDIVIDUAL
KARL H. SHINGLER

22b. TELEPHONE NUMBER *(Include Area Code)*
412 268-7630

22c. OFFICE SYMBOL
SEI JPO

EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE