What a software engineer needs to know.

Mary Shaw  
*Carnegie Mellon University*

Dario Giuse

Raj Reddy

Follow this and additional works at: [http://repository.cmu.edu/compsci](http://repository.cmu.edu/compsci)
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
What A Software Engineer Needs to Know:
I. Program Vocabulary

Mary Shaw, Dario Giuse, Raj Reddy
22 August 1989
CMU-CS-89-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213


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.

© 1989 Mary Shaw, Dario Giuse, Raj Reddy

This research was 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, 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 49433-6543. M. Shaw was also sponsored in part by the Software Engineering Institute under contract to the Department of Defense.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
# Table of Contents

1. Proficiency Requires Content Knowledge  
   1.1. The Magic Number $70,000 \pm 20,000$  
   1.2. The Content Component of Fluency  
   1.3. The Incremental Nature of Vocabulary  
      1.3.1. General Vocabulary  
      1.3.2. Technical Vocabulary  
      1.3.3. Growth of General and Technical Vocabulary  
   1.4. Engineering Design Requires Content Knowledge  
   1.5. Software Development Requires Content Knowledge  
2. Empirical Observations: Vocabularies of 31 Programs  
   2.1. Approach  
   2.2. Summary  
   2.3. Details for Lisp Programs  
   2.4. Details for C Programs  
   2.5. Details for Ada Programs  
3. Implications for Education and Practice  
   3.1. Memory for Vocabulary Acquisition  
      3.1.1 How Vocabulary is Learned  
      3.1.2 Recommendations: Read a Good Program, Write a Good Program  
   3.2. Reference Materials and Tools for Access  
      3.2.1. The Role of Reference Materials  
      3.2.2. Recognition Vocabulary Supports Maintenance  
      3.2.3. Generation Vocabulary is Needed for Development  
      3.2.4. Recommendations: Know Where (and How) to Look It Up  
   3.3. Derivation of Meaning from Context  
      3.3.1. Recommendations: When in Rome, Talk As the Romans Do  
   3.4. Implications for Productivity  
      3.4.1. Recommendations: Try It, You'll Like It  
4. References
List of Figures

Figure 1: Vocabulary Families 3
Figure 2: Overall Distribution of Symbols in Vocabulary and Usage 8
Figure 3: Lisp Vocabulary 12
Figure 4: Lisp Usage 12
Figure 5: C Vocabulary 15
Figure 6: C Usage 16
Figure 7: Ada Vocabulary 19
Figure 8: Ada Usage 20
List of Tables

Table 1: Summary Statistics
Table 2: Learnable Vocabulary
Table 3: Uses per Symbol in Learnable Vocabulary
Table 4: Statistics on Lisp Programs
Table 5: Statistics on C Programs
Table 6: Statistics on Ada Programs
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 English 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.

<table>
<thead>
<tr>
<th>Number of systems</th>
<th>Lisp</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total size (lines)</td>
<td>68,415</td>
<td>47,991</td>
<td>67,945</td>
</tr>
<tr>
<td>Average size (lines)</td>
<td>8,552</td>
<td>4,363</td>
<td>5,662</td>
</tr>
<tr>
<td>Range (lines)</td>
<td>317-40,539</td>
<td>926-11,240</td>
<td>24-19,648</td>
</tr>
<tr>
<td>Average size (# symbols)</td>
<td>19,435</td>
<td>13,623</td>
<td>12,429</td>
</tr>
<tr>
<td>Average vocabulary (# distinct symbols)</td>
<td>2,245</td>
<td>820</td>
<td>715</td>
</tr>
<tr>
<td>Average usage rate (size/vocabulary)</td>
<td>8.7</td>
<td>16.5</td>
<td>17.4</td>
</tr>
<tr>
<td>Distinct instances (vocabulary size)</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Built-in symbols</td>
<td>469</td>
<td>64</td>
<td>115</td>
</tr>
<tr>
<td>Common library symbols</td>
<td>498</td>
<td>256</td>
<td>28</td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>2,693</td>
<td>1,414</td>
<td>2,394</td>
</tr>
<tr>
<td>Literals</td>
<td>7,394</td>
<td>2,640</td>
<td>1,030</td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>4,434</td>
<td>2,824</td>
<td>2,404</td>
</tr>
<tr>
<td>Occurrences in the program</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Built-in symbols</td>
<td>46,208</td>
<td>53,546</td>
<td>62,559</td>
</tr>
<tr>
<td>Common library symbols</td>
<td>7,227</td>
<td>6,429</td>
<td>4,492</td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>23,342</td>
<td>10,785</td>
<td>34,353</td>
</tr>
<tr>
<td>Literals</td>
<td>28,106</td>
<td>20,778</td>
<td>11,600</td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>50,598</td>
<td>58,311</td>
<td>36,147</td>
</tr>
</tbody>
</table>

Table 1: Summary Statistics
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.

<table>
<thead>
<tr>
<th></th>
<th>Lisp</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>3,660</td>
<td>1,734</td>
<td>2,537</td>
</tr>
<tr>
<td></td>
<td>76,777</td>
<td>70,760</td>
<td>101,404</td>
</tr>
</tbody>
</table>

Within the learnable vocabulary:

<table>
<thead>
<tr>
<th></th>
<th>Lisp</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Built-in</td>
<td>12.8%</td>
<td>3.7%</td>
<td>4.5%</td>
</tr>
<tr>
<td>% Common library</td>
<td>13.6%</td>
<td>14.8%</td>
<td>1.1%</td>
</tr>
<tr>
<td>% System-specific</td>
<td>73.6%</td>
<td>81.5%</td>
<td>94.4%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>Lisp</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in</td>
<td>4.7</td>
<td>20.6</td>
<td>13.5</td>
</tr>
<tr>
<td>Common library</td>
<td>0.7</td>
<td>0.6</td>
<td>4.0</td>
</tr>
<tr>
<td>System-specific</td>
<td>0.4</td>
<td>0.2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th></th>
<th>hemlock.lisp</th>
<th>inter.lisp</th>
<th>kr.lisp</th>
<th>lapidary.lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (lines)</td>
<td>40,539</td>
<td>5,197</td>
<td>2,419</td>
<td>11,400</td>
</tr>
<tr>
<td>Size (# lexical tokens)</td>
<td>89,851</td>
<td>13,063</td>
<td>3,928</td>
<td>23,559</td>
</tr>
<tr>
<td>Vocab (# distinct tokens)</td>
<td>10,034</td>
<td>2,121</td>
<td>516</td>
<td>2,221</td>
</tr>
</tbody>
</table>

**Distinct instances (vocabulary size)**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in symbols</td>
<td>410</td>
<td>4.1%</td>
<td>135</td>
<td>6.4%</td>
<td>134</td>
<td>26.0%</td>
<td>151</td>
<td>6.8%</td>
</tr>
<tr>
<td>Common library symbols</td>
<td>307</td>
<td>3.1%</td>
<td>97</td>
<td>4.6%</td>
<td>0</td>
<td>0.0%</td>
<td>107</td>
<td>4.8%</td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>1,502</td>
<td>15.0%</td>
<td>190</td>
<td>9.0%</td>
<td>101</td>
<td>19.6%</td>
<td>293</td>
<td>13.2%</td>
</tr>
<tr>
<td>Literals</td>
<td>5,188</td>
<td>51.7%</td>
<td>1,232</td>
<td>58.1%</td>
<td>120</td>
<td>23.3%</td>
<td>1,071</td>
<td>48.2%</td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>2,627</td>
<td>26.2%</td>
<td>467</td>
<td>22.0%</td>
<td>161</td>
<td>31.2%</td>
<td>599</td>
<td>27.0%</td>
</tr>
</tbody>
</table>

**Occurrences in the program**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in symbols</td>
<td>27,524</td>
<td>30.6%</td>
<td>3,327</td>
<td>25.5%</td>
<td>1,566</td>
<td>39.9%</td>
<td>5,709</td>
<td>23.8%</td>
</tr>
<tr>
<td>Common library symbols</td>
<td>3,327</td>
<td>9.9%</td>
<td>1,299</td>
<td>9.9%</td>
<td>0</td>
<td>0.0%</td>
<td>4,132</td>
<td>17.2%</td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>828</td>
<td>6.3%</td>
<td>586</td>
<td>14.9%</td>
<td>977</td>
<td>4.1%</td>
<td>11</td>
<td>0.0%</td>
</tr>
<tr>
<td>Literals</td>
<td>3,646</td>
<td>27.9%</td>
<td>269</td>
<td>6.8%</td>
<td>7,301</td>
<td>30.5%</td>
<td>17</td>
<td>0.0%</td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>3,963</td>
<td>30.3%</td>
<td>161</td>
<td>21.1%</td>
<td>599</td>
<td>28.5%</td>
<td>252</td>
<td>23.2%</td>
</tr>
</tbody>
</table>

**Table 4: Statistics on Lisp programs**

<table>
<thead>
<tr>
<th></th>
<th>opal.lisp</th>
<th>ops.lisp</th>
<th>profile.lisp</th>
<th>psgraph.lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (lines)</td>
<td>3,594</td>
<td>4,457</td>
<td>317</td>
<td>492</td>
</tr>
<tr>
<td>Size (# lexical tokens)</td>
<td>9,323</td>
<td>13,488</td>
<td>551</td>
<td>1,319</td>
</tr>
<tr>
<td>Vocab (# distinct tokens)</td>
<td>1,087</td>
<td>1,525</td>
<td>130</td>
<td>293</td>
</tr>
</tbody>
</table>

**Distinct instances (vocabulary size)**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in symbols</td>
<td>154</td>
<td>14.2%</td>
<td>153</td>
<td>10.0%</td>
<td>79</td>
<td>41.6%</td>
<td>51</td>
<td>19.4%</td>
</tr>
<tr>
<td>Common library symbols</td>
<td>98</td>
<td>9.0%</td>
<td>0</td>
<td>0.0%</td>
<td>11</td>
<td>5.8%</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>218</td>
<td>20.1%</td>
<td>382</td>
<td>25.0%</td>
<td>16</td>
<td>8.4%</td>
<td>6</td>
<td>2.3%</td>
</tr>
<tr>
<td>Literals</td>
<td>365</td>
<td>33.6%</td>
<td>266</td>
<td>17.4%</td>
<td>44</td>
<td>23.2%</td>
<td>131</td>
<td>49.8%</td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>252</td>
<td>23.2%</td>
<td>724</td>
<td>47.5%</td>
<td>40</td>
<td>21.1%</td>
<td>75</td>
<td>28.5%</td>
</tr>
</tbody>
</table>

**Occurrences in the program**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in symbols</td>
<td>2,239</td>
<td>24.0%</td>
<td>5,028</td>
<td>37.3%</td>
<td>262</td>
<td>47.5%</td>
<td>553</td>
<td>42.0%</td>
</tr>
<tr>
<td>Common library symbols</td>
<td>946</td>
<td>10.1%</td>
<td>0</td>
<td>0.0%</td>
<td>15</td>
<td>2.7%</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>870</td>
<td>9.3%</td>
<td>1,598</td>
<td>11.8%</td>
<td>45</td>
<td>8.2%</td>
<td>17</td>
<td>1.3%</td>
</tr>
<tr>
<td>Literals</td>
<td>2,167</td>
<td>23.2%</td>
<td>806</td>
<td>6.0%</td>
<td>83</td>
<td>15.1%</td>
<td>187</td>
<td>14.2%</td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>3,101</td>
<td>33.3%</td>
<td>6,056</td>
<td>44.9%</td>
<td>146</td>
<td>26.5%</td>
<td>561</td>
<td>42.6%</td>
</tr>
</tbody>
</table>
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.
<table>
<thead>
<tr>
<th></th>
<th>avietest.c</th>
<th>chinese.c</th>
<th>dynload.c</th>
<th>fscript.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (lines)</td>
<td>4,187</td>
<td>5,248</td>
<td>1,328</td>
<td>9,867</td>
</tr>
<tr>
<td>Size (# lexical tokens)</td>
<td>14,401</td>
<td>12,249</td>
<td>357</td>
<td>43,590</td>
</tr>
<tr>
<td>Vocab (# distinct tokens)</td>
<td>513</td>
<td>1,395</td>
<td>425</td>
<td>1,930</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISTINCT INSTANCES</th>
<th>Built-in symbols</th>
<th>Common library symbols</th>
<th>System-specific symbols</th>
<th>Literals</th>
<th>Purely local symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>46</td>
<td>45</td>
<td>58</td>
<td>167</td>
<td>197</td>
</tr>
<tr>
<td>%</td>
<td>9.0%</td>
<td>8.8%</td>
<td>11.3%</td>
<td>32.5%</td>
<td>39.4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OCCURRENCES IN THE PROGRAM</th>
<th>Built-in symbols</th>
<th>Common library symbols</th>
<th>System-specific symbols</th>
<th>Literals</th>
<th>Purely local symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>5,764</td>
<td>1,413</td>
<td>299</td>
<td>687</td>
<td>6,238</td>
</tr>
<tr>
<td>%</td>
<td>40.0%</td>
<td>9.8%</td>
<td>2.1%</td>
<td>4.8%</td>
<td>43.3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>mololo.c</th>
<th>rfr.c</th>
<th>sc.c</th>
<th>descartes.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (lines)</td>
<td>11,240</td>
<td>1,199</td>
<td>2,598</td>
<td>7,928</td>
</tr>
<tr>
<td>Size (# lexical tokens)</td>
<td>40,082</td>
<td>4,103</td>
<td>5,583</td>
<td>15,538</td>
</tr>
<tr>
<td>Vocab (# distinct tokens)</td>
<td>1,376</td>
<td>341</td>
<td>516</td>
<td>1,377</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISTINCT INSTANCES</th>
<th>Built-in symbols</th>
<th>Common library symbols</th>
<th>System-specific symbols</th>
<th>Literals</th>
<th>Purely local symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>45</td>
<td>24</td>
<td>173</td>
<td>492</td>
<td>642</td>
</tr>
<tr>
<td>%</td>
<td>3.3%</td>
<td>1.7%</td>
<td>12.6%</td>
<td>35.8%</td>
<td>46.7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OCCURRENCES IN THE PROGRAM</th>
<th>Built-in symbols</th>
<th>Common library symbols</th>
<th>System-specific symbols</th>
<th>Literals</th>
<th>Purely local symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>15,304</td>
<td>1,459</td>
<td>664</td>
<td>5,410</td>
<td>17,245</td>
</tr>
<tr>
<td>%</td>
<td>38.2%</td>
<td>3.6%</td>
<td>1.7%</td>
<td>13.5%</td>
<td>43.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>airlog.c</th>
<th>ckbook.c</th>
<th>crostic.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (lines)</td>
<td>2,344</td>
<td>926</td>
<td>1,125</td>
</tr>
<tr>
<td>Size (# lexical tokens)</td>
<td>5,278</td>
<td>2,574</td>
<td>2,594</td>
</tr>
<tr>
<td>Vocab (# distinct tokens)</td>
<td>563</td>
<td>222</td>
<td>403</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISTINCT INSTANCES</th>
<th>Built-in symbols</th>
<th>Common library symbols</th>
<th>System-specific symbols</th>
<th>Literals</th>
<th>Purely local symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>35</td>
<td>68</td>
<td>102</td>
<td>220</td>
<td>138</td>
</tr>
<tr>
<td>%</td>
<td>6.2%</td>
<td>12.1%</td>
<td>18.1%</td>
<td>39.1%</td>
<td>24.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OCCURRENCES IN THE PROGRAM</th>
<th>Built-in symbols</th>
<th>Common library symbols</th>
<th>System-specific symbols</th>
<th>Literals</th>
<th>Purely local symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>1,571</td>
<td>632</td>
<td>622</td>
<td>581</td>
<td>1,872</td>
</tr>
<tr>
<td>%</td>
<td>29.8%</td>
<td>12.0%</td>
<td>11.8%</td>
<td>11.0%</td>
<td>35.5%</td>
</tr>
</tbody>
</table>

Table 5: Statistics on C Programs
Figure 5: C Vocabulary
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.

<table>
<thead>
<tr>
<th></th>
<th>abstractions.ada</th>
<th>ada-sql.ada</th>
<th>alsptypes.ada</th>
<th>expert.ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (lines)</td>
<td>1,781</td>
<td>19,648</td>
<td>6,273</td>
<td>1,048</td>
</tr>
<tr>
<td>Size (# lexical tokens)</td>
<td>25,883</td>
<td>51,626</td>
<td>12,203</td>
<td>2,127</td>
</tr>
<tr>
<td>Vocab (# distinct tokens)</td>
<td>1,484</td>
<td>2,066</td>
<td>603</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distinct Instances (vocabulary size)</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in symbols</td>
<td>85</td>
<td>5.7%</td>
<td>87</td>
<td>4.2%</td>
<td>65</td>
<td>10.8%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common library symbols</td>
<td>20</td>
<td>1.3%</td>
<td>21</td>
<td>1.0%</td>
<td>10</td>
<td>1.7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>658</td>
<td>44.3%</td>
<td>842</td>
<td>40.8%</td>
<td>140</td>
<td>23.2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Literals</td>
<td>227</td>
<td>15.3%</td>
<td>109</td>
<td>5.3%</td>
<td>130</td>
<td>21.6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>494</td>
<td>33.3%</td>
<td>1,007</td>
<td>48.7%</td>
<td>258</td>
<td>42.8%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Occurrences in the program</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
<th>N</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in symbols</td>
<td>11,941</td>
<td>46.1%</td>
<td>21,544</td>
<td>41.7%</td>
<td>4,161</td>
<td>34.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common library symbols</td>
<td>247</td>
<td>1.0%</td>
<td>713</td>
<td>1.4%</td>
<td>781</td>
<td>6.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System-specific symbols</td>
<td>7,117</td>
<td>27.5%</td>
<td>13,349</td>
<td>25.9%</td>
<td>3,342</td>
<td>27.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Literals</td>
<td>1,050</td>
<td>4.1%</td>
<td>3,546</td>
<td>6.9%</td>
<td>406</td>
<td>3.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purely local symbols</td>
<td>5,528</td>
<td>21.4%</td>
<td>12,474</td>
<td>24.2%</td>
<td>3,511</td>
<td>28.8%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 how to use the knowledge, and the direct cost each time the knowledge is used in practice. These costs have different distributions:

<table>
<thead>
<tr>
<th></th>
<th>Infrastructure Cost</th>
<th>Initial Learning Cost</th>
<th>Cost of Use in Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Reference</td>
<td>high</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>Derivation</td>
<td>medium-high</td>
<td>medium</td>
<td>high</td>
</tr>
</tbody>
</table>

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


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.

Raj Reddy.
"Foundations and Grand Challenges of Artificial Intelligence."
(1988 presidential address, American Association for Artificial Intelligence)

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.

G. L. Steele.
Common LISP—The Language.

Warren Teitelman.
"A tour through Cedar."

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

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