The New C Standard (Usage, Figures, and Tables)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

CHANGES

Copyright © 2005, 2008 Derek Jones -5
The material in the C99 subsections is copyright © ISO. The material in the C90 and C++ sections that is quoted from the respective language standards is copyright © ISO.
Credits and permissions for quoted material is given where that material appears.
THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE PARTICULAR WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.
THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN.

Commentary

The phrase *at the time of writing* is sometimes used. For this version of the material this time should be taken to mean no later than December 2008.

```
29 Jan 2008  1.1   Integrated in changes made by TC3, required C sentence renumbering.
                   60+ recent references added + associated commentary.
                   A few Usage figures and tables added.
                   Page layout improvements.  Lots of grammar fixes.
 5 Aug 2005  1.0b  Many hyperlinks added.  pdf searching through page 782 speeded up.
                   Various typos fixed (over 70% reported by Tom Plum).
16 Jun 2005  1.0a  Improvements to character set discussion (thanks to Kent Karlsson), margin
                   references, C99 footnote number typos, and various other typos fixed.
30 May 2005  1.0   Initial release.
```

0 With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.26]) is discouraged.

This International Standard is divided into four major subdivisions:

— preliminary elements (clauses 1–4);

— the characteristics of environments that translate and execute C programs (clause 5);

— the language syntax, constraints, and semantics (clause 6);

— the library facilities (clause 7).

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.

The language clause (clause 6) is derived from “The C Reference Manual”.

The library clause (clause 7) is based on the 1984 */usr/group Standard*.

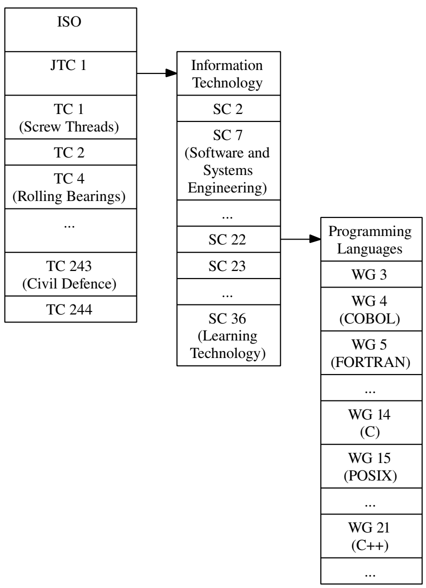


Figure 0.1: The ISO Technical Committee structure— JTC (Joint Technical Committee, with the IEC in this case), TC (Technical Committee), SC (Standards Committee), WG (Working Group).

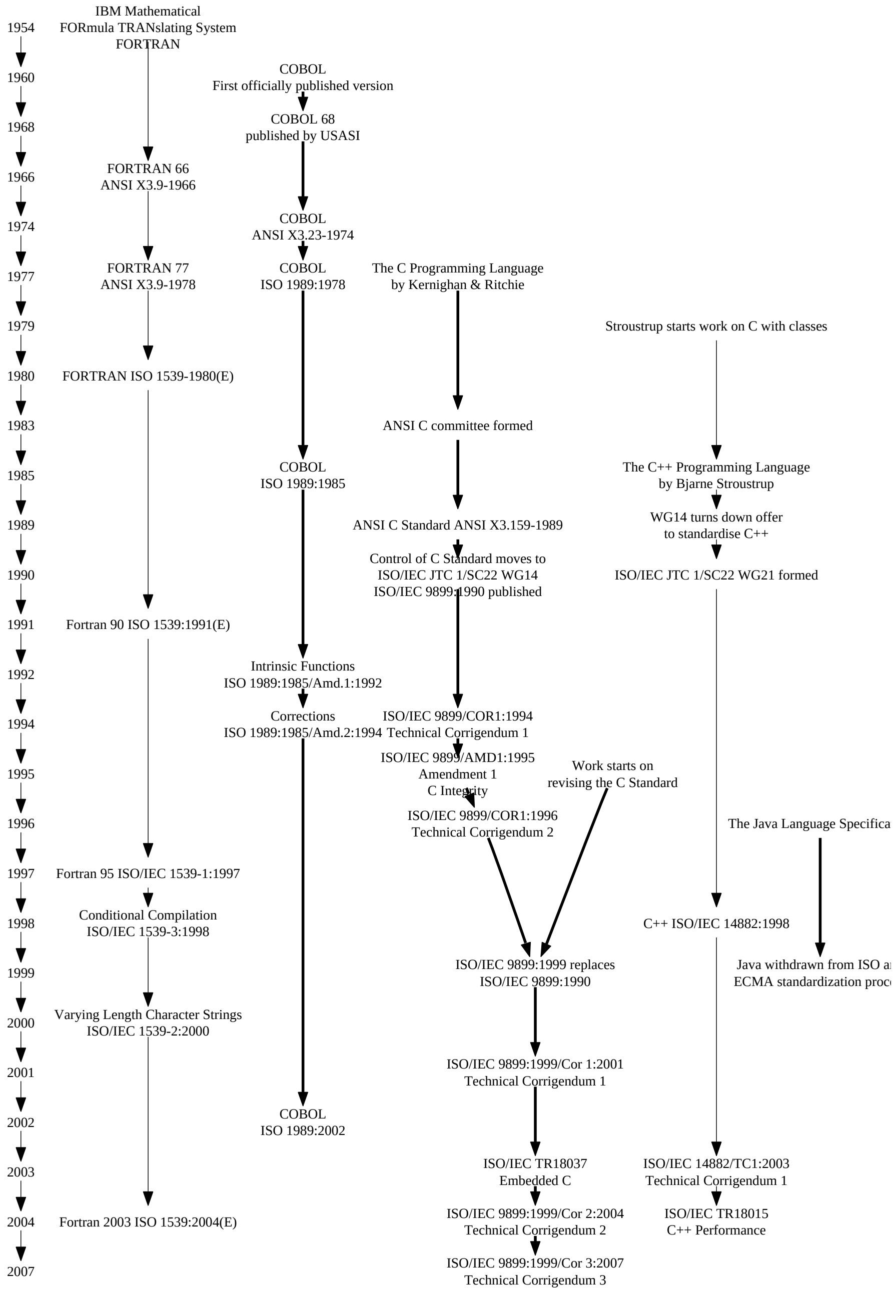


Figure 0.2: Outline history of the C language and a few long-lived languages. (Backus[8] describes the earliest history of Fortran.)

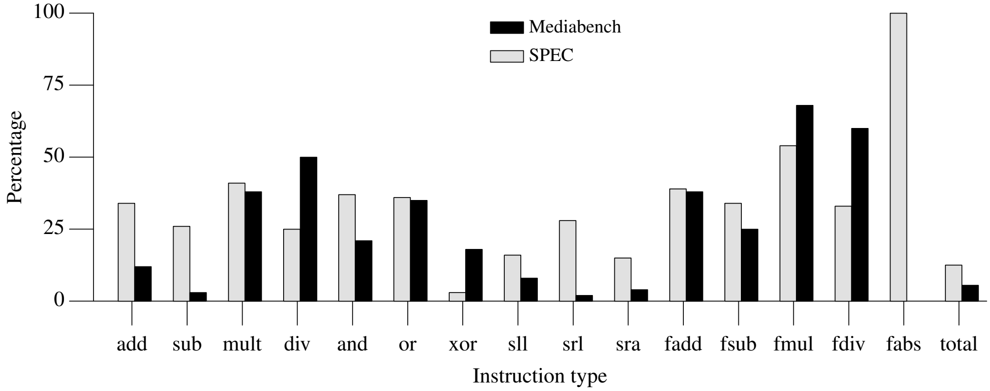


Figure 0.3: Dynamic frequency, percentage calculated over shown instructions (last column gives percentage of these instruction relative to all instructions executed) during execution of the SPEC and MediaBench benchmarks of some computational oriented instructions. Adapted from Yi and Lilja.[175]

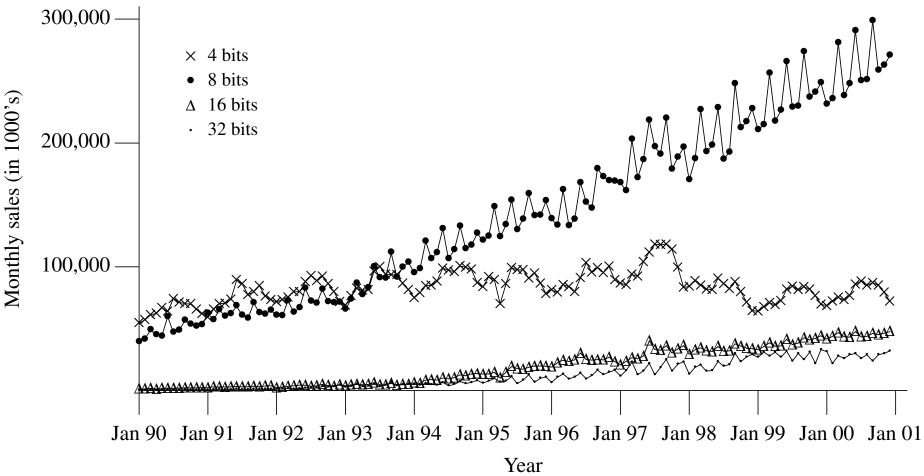


Figure 0.4: Monthly unit sales of microprocessors having a given bus width. Adapted from Turley[168] (using data supplied by Turley).

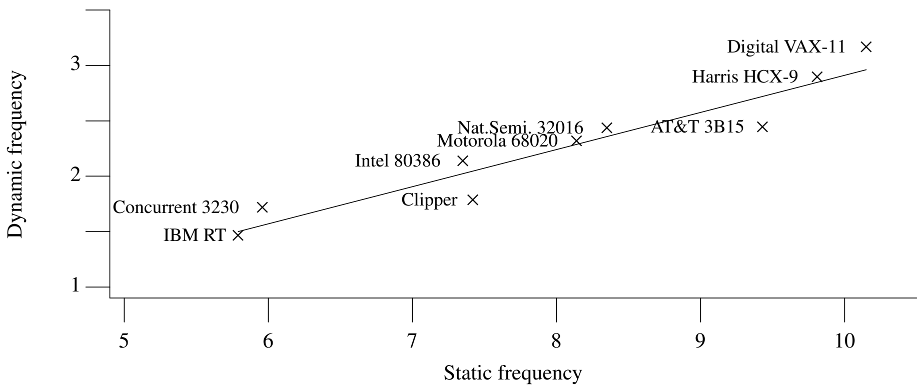


Figure 0.5: Dynamic/static frequency of *call* instructions. Adapted from Davidson.[51]

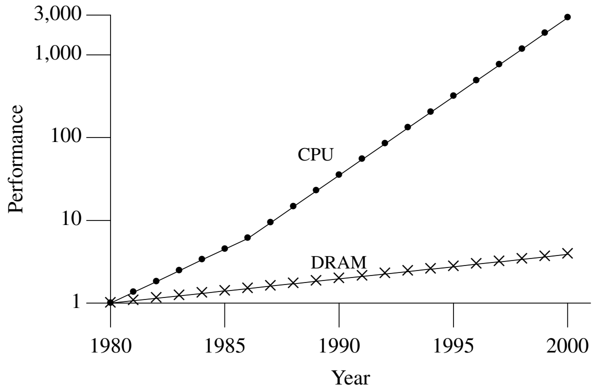


Figure 0.6: Relative performance of CPU against storage (DRAM), 1980==1. Adapted from Hennessy.[80]

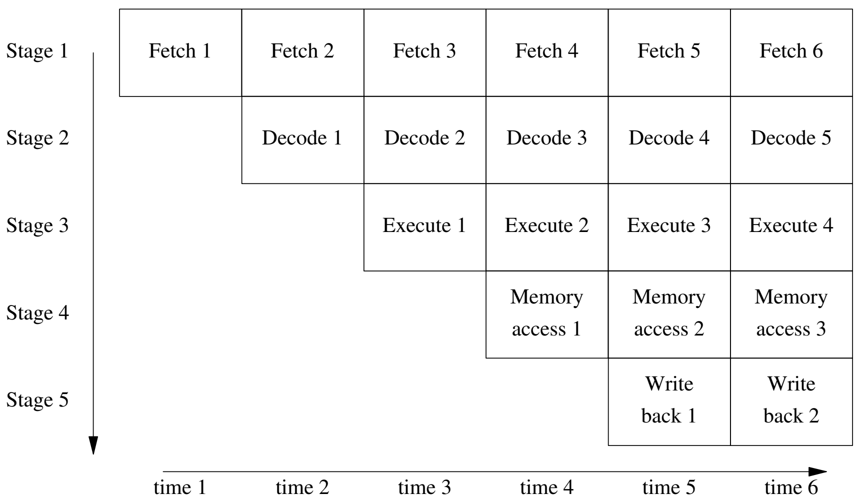


Figure 0.7: Simplified diagram of some typical stages in a processor instruction pipeline: Instruction fetch, decode, execute, memory access, and write back.

Table 0.1: Percentage of reported problems having a given mean time to first problem occurrence (in months, summed over all installations of a product) for various products (numbered 1 to 9), e.g., 28.8% of the reported faults in product 1 were, on average, first reported after 19,000 months of program execution time (another 34.2% of problems were first reported after 60,000 months). From Adams.[2]

Product	19	60	190	600	1,900	6,000	19,000	60,000
1	0.7	1.2	2.1	5.0	10.3	17.8	28.8	34.2
2	0.7	1.5	3.2	4.5	9.7	18.2	28.0	34.3
3	0.4	1.4	2.8	6.5	8.7	18.0	28.5	33.7
4	0.1	0.3	2.0	4.4	11.9	18.7	28.5	34.2
5	0.7	1.4	2.9	4.4	9.4	18.4	28.5	34.2
6	0.3	0.8	2.1	5.0	11.5	20.1	28.2	32.0
7	0.6	1.4	2.7	4.5	9.9	18.5	28.5	34.0
8	1.1	1.4	2.7	6.5	11.1	18.4	27.1	31.9
9	0.0	0.5	1.9	5.6	12.8	20.4	27.6	31.2

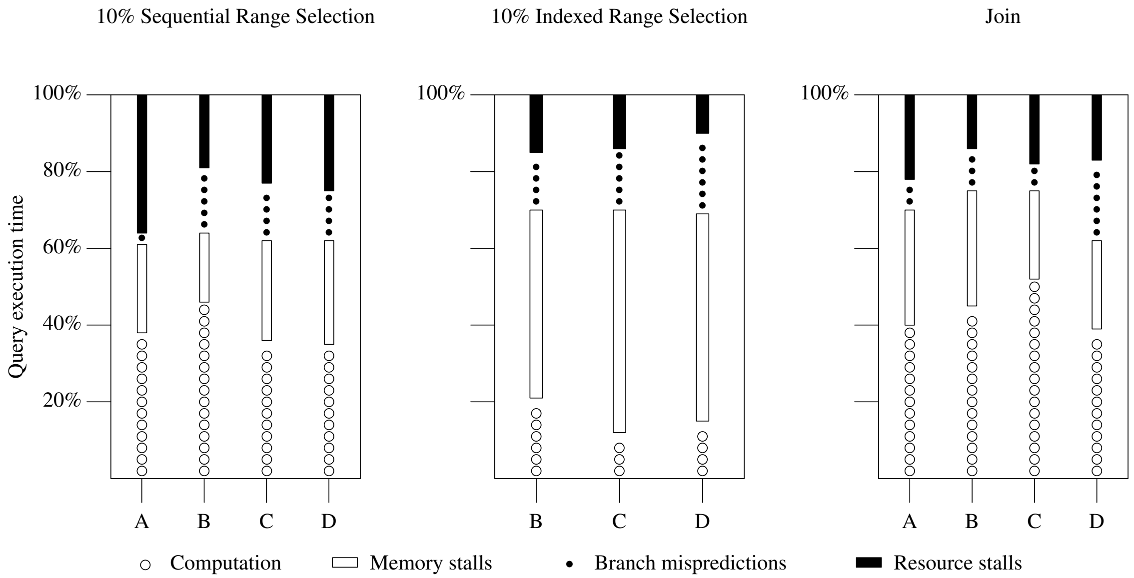


Figure 0.8: Execution time breakdown, by four processor components (bottom of graphs) for three different application queries (top of graphs). Adapted from Ailamaki.[3]

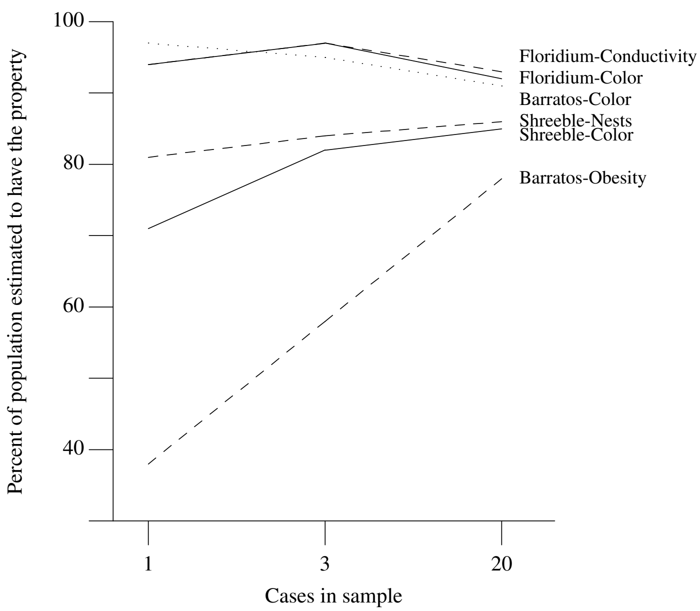


Figure 0.9: Percentage of population estimated to have the sample property against the number of cases in the sample. Adapted from Nisbett.[130]

Table 0.2: Fault categories ordered by frequency of occurrence. The last column is the rank position after the fault fix weighting factor is taken into account. Based on Perry.[138]

Rank	Fault Description	% Total Faults	Fix Rank	Rank	Fault Description	% Total Faults	Fix Rank
1	internal functionality	25.0	13	12	error handling	3.3	6
2	interface complexity	11.4	10	13	primitive's misuse	2.4	11
3	unexpected dependencies	8.0	4	14	dynamic data use	2.1	15
4	low-level logic	7.9	17	15	resource allocation	1.5	2
5	design/code complexity	7.7	3	16	static data design	1.0	19
6	other	5.8	12	17	performance	0.9	1
7	change coordinates	4.9	14	18	unknown interactions	0.7	5
8	concurrent work	4.4	9	19	primitives unsupported	0.6	19
9	race conditions	4.3	7	20	IPC rule violated	0.4	16
10	external functionality	3.6	8	21	change management complexity	0.3	21
11	language pitfalls i.e., use of = when == intended	3.5	18	22	dynamic data design	0.3	21

Table 0.3: Underlying cause of faults. The *none given* category occurs because sometimes both the fault and the underlying cause are the same. For instance, *language pitfalls*, or *low-level logic*. Based on Perry.[138]

Rank	Cause Description	% Total Causes	Fix Rank
1	Incomplete/omitted design	25.2	3
2	None given	20.5	10
3	Lack of knowledge	17.8	8
4	Ambiguous design	9.8	9
5	Earlier incorrect fix	7.3	7
6	Submitted under duress	6.8	6
7	Incomplete/omitted requirements	5.4	2
8	Other	4.1	4
9	Ambiguous requirements	2.0	1
10	Incorrect modifications	1.1	5

Table 0.4: Means of fault prevention. The last column is the rank position after the fault fix weighting factor is taken into account. Based on Perry.[138]

Rank	Means Description	% Observed	Fix Rank
1	Application walk-through	24.5	8
2	Provide expert/clearer documentation	15.7	3
3	Guideline enforcement	13.3	10
4	Requirements/design templates	10.0	5
5	Better test planning	9.9	9
6	Formal requirements	8.8	2
7	Formal interface specifications	7.2	4
8	Other	6.9	6
9	Training	2.2	1
10	Keep document/code in sync	1.5	7

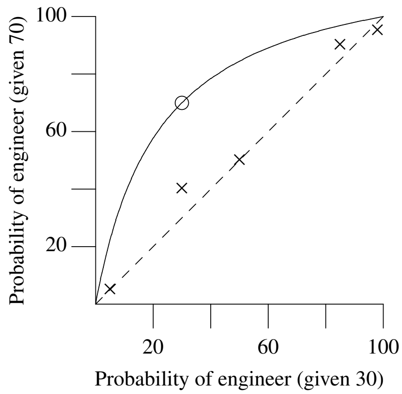


Figure 0.10: Median judged probability of subjects choosing an engineer, for five descriptions and for the null description (unfilled circle symbol). Adapted from Kahneman.[90]

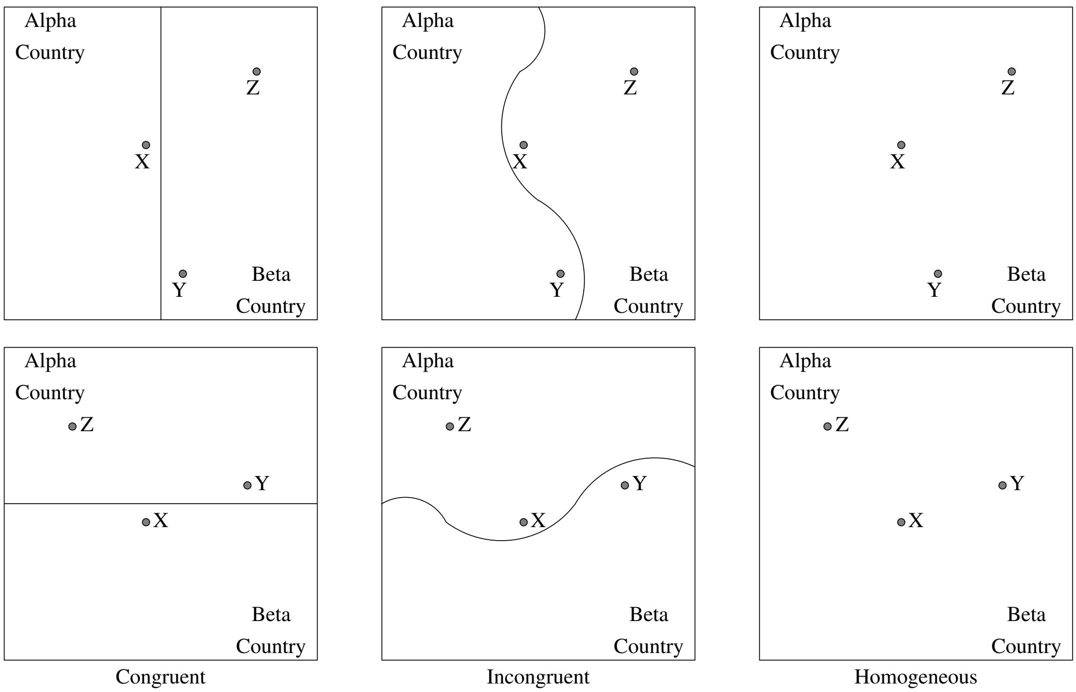


Figure 0.11: Country boundaries distort judgment of relative city locations. Adapted from Stevens.[158]

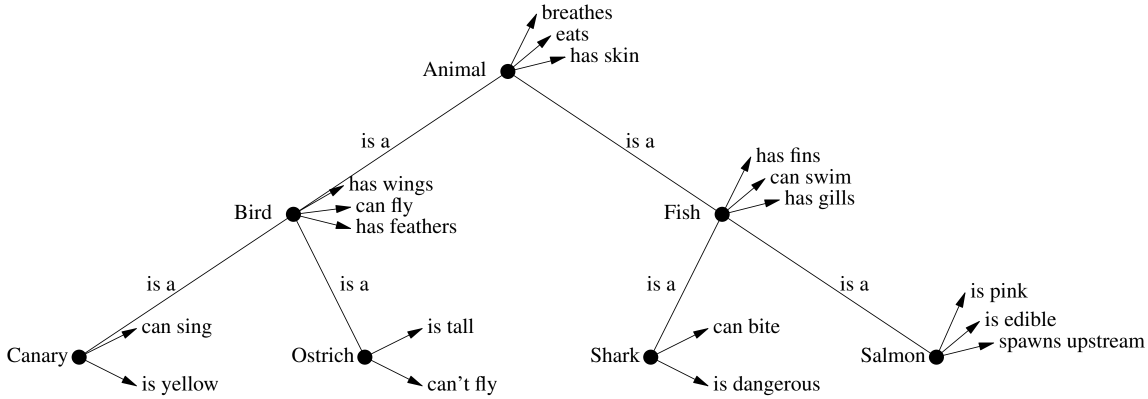


Figure 0.12: Hypothetical memory structure for a three-level hierarchy. Adapted from Collins.[43]

Table 0.5: General properties of explanations and their potential role in understanding conceptual coherence. Adapted from Murphy.[126]

Properties of Explanations	Role in Conceptual Coherence
Explanation of a sort, specified over some domain of observation	Constrains which attributes will be included in a concept representation Focuses on certain relationships over others in detecting attribute correlations
Simplify reality	Concepts may be idealizations that impose more structure than is *objectively* present
Have an external structure— fits in with (or do not contradict) what is already known	Stresses intercategory structure; attributes are considered essential to the degree that they play a part in related theories (external structures)
Have an internal structure— defined in part by relations connecting attributes	Emphasizes mutual constraints among attributes. May suggest how concept attributes are learned
Interact with data and observations in some way	Calls attention to inference processes in categorization and suggests that more than attribute matching is involved

Table 0.6: Computation of pattern similarity. Adapted from Estes.[64]

Attribute	1	2	3	4	5	6
Starling	+	+	-	+	+	+
Sandpiper	+	+	+	+	-	+
Attribute similarity	t	t	s_3	t	s_5	t

Table 0.7: Computation of similarity to category. Adapted from Estes.[64]

Object	Ro	Bl	Sw	St	Vu	Sa	Ch	Fl	Pe	Similarity to Category
Robin	1	1	1	s	s^4	s	s^5	s^6	s^5	$3+2s+s^4+2s^5+s^6$
Bluebird	1	1	1	s	s^4	s	s^5	s^6	s^5	$3+2s+s^4+2s^5+s^6$
Swallow	1	1	1	s	s^4	s	s^5	s^6	s^5	$3+2s+s^4+2s^5+s^6$
Starling	s	s	s	1	s^3	s^2	s^6	s^5	s^6	$1+3s+s^2+s^3+s^5+2s^6$
Vulture	s^4	s^4	s^4	s^3	1	s^5	s^3	s^2	s^3	$1+s^2+3s^3+3s^4+s^5$
Sandpiper	s	s	s	s^2	s^5	1	s^4	s^5	s^4	$1+3s+s^2+s^4+s^5$
Chicken	s^5	s^5	s^5	s^6	s^3	s^4	1	s	1	$2+s+s^3+s^4+3s^5+s^6$
Flamingo	s^6	s^6	s^6	s^5	s^2	s^5	s	1	s	$1+2s+s^2+2s^5+3s^6$
Penguin	s^5	s^5	s^5	s^6	s^3	s^4	1	s	1	$2+s+s^3+s^4+3s^5+s^6$

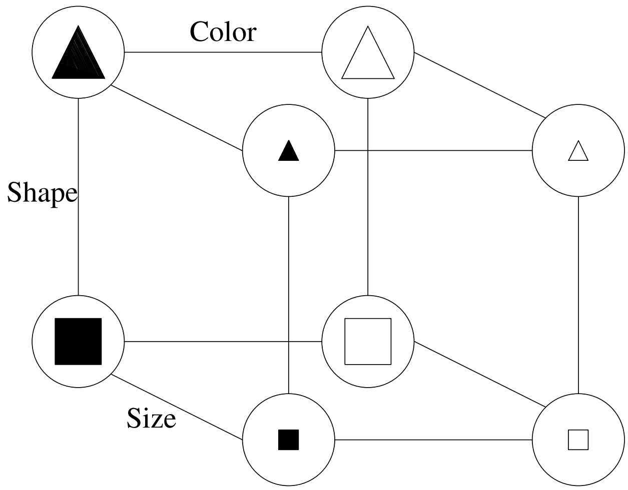


Figure 0.13: Representation of stimuli with shape in the horizontal plane and color in one of the vertical planes. Adapted from Shepard.[149]

Table 0.8: Computation of weighted similarity to category. From Estes.[64]

Object	Similarity Formula	$s = 0.5$	Relative Frequency	Weighted Similarity
Robin	$3 + 2s + s^4 + 2s^5 + s^6$	4.14	0.30	1.24
Bluebird	$3 + 2s + s^4 + 2s^5 + s^6$	4.14	0.20	0.83
Swallow	$3 + 2s + s^4 + 2s^5 + s^6$	4.14	0.10	0.41
Starling	$1 + 3s + s^2 + s^3 + s^5 + 2s^6$	2.94	0.15	0.44
Vulture	$1 + s^2 + 3s^3 + 3s^4 + s^5$	1.84	0.02	0.04
Sandpiper	$1 + 3s + s^2 + s^4 + s^5$	2.94	0.05	0.15
Chicken	$2 + s + s^3 + s^4 + 3s^5 + s^6$	2.80	0.15	0.42
Flamingo	$1 + 2s + s^2 + 2s^5 + 3s^6$	2.36	0.01	0.02
Penguin	$2 + s + s^3 + s^4 + 3s^5 + s^6$	2.80	0.02	0.06

Table 0.9: Similarity to category (black triangle and black square assigned to category A; white triangle and white square assigned to category B).

Stimulus	Similarity to A	Similarity to B
Dark triangle	$1 + s$	$s + s^2$
Dark square	$1 + s$	$s + s^2$
Light triangle	$s + s^2$	$1 + s$
Light square	$s + s^2$	$1 + s$

Table 0.10: Similarity to category (black triangle and white square assigned to category A; white triangle and black square assigned to category B).

Stimulus	Similarity to A	Similarity to B
Dark triangle	$s + s^2$	$2s$
Dark square	$2s$	$s + s^2$
Light triangle	$2s$	$s + s^2$
Light square	$s + s^2$	$2s$

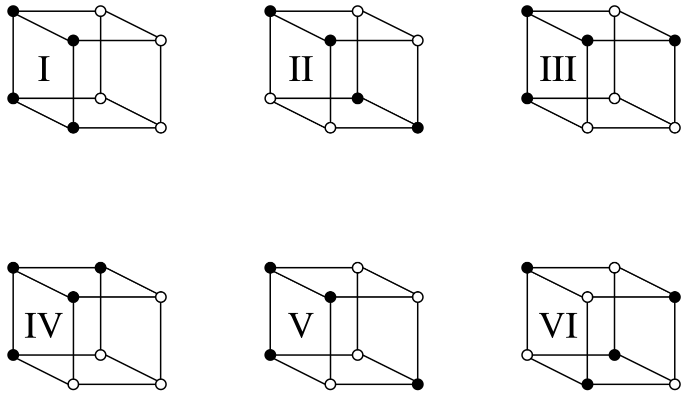


Figure 0.14: One of the six unique configurations (i.e., it is not possible to rotate one configuration into another within the set of six) of selecting four times from eight possibilities. Adapted from Shepard.[149]

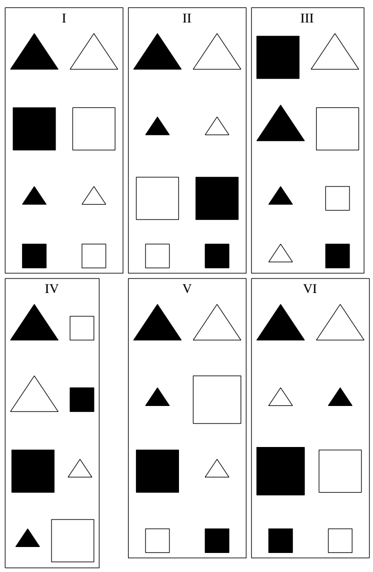


Figure 0.15: Example list of categories. Adapted from Shepard.[149]

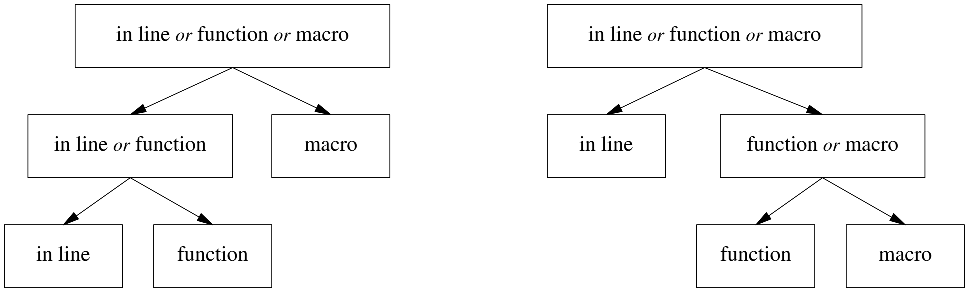


Figure 0.16: Possible decision paths when making pair-wise comparisons on whether to use a inline code, a function, or a macro; for two different pair-wise associations.

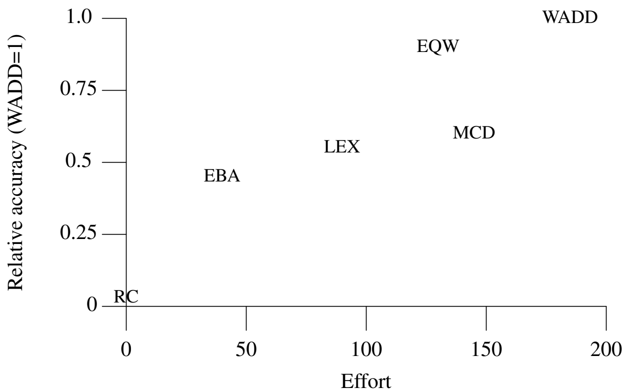


Figure 0.17: Effort and accuracy levels for various decision-making strategies; EBA (Elimination-by-aspects heuristic), EQW (equal weight heuristic), LEX (lexicographic heuristic), MCD (majority of confirming dimensions heuristic), RC (Random choice), and WADD (weighted additive rule). Adapted from Payne.[135]

Table 0.11: Storage/Execution performance alternatives.

Alternative	Storage Needed	Speed of Execution
X	7 K	Low
Y	15 K	High
Z	10 K	Medium

Table 0.12: Inducement of intuitive cognition and analytic cognition, by task conditions. Adapted from Hammond.[76]

Task Characteristic	Intuition-Inducing State of Task Characteristic	Analysis-Inducing State of Task Characteristic
Number of cues	Large (>5)	Small
Measurement of cues	Perceptual measurement	Objective reliable measurement
Distribution of cue values	Continuous highly variable distribution	Unknown distribution; cues are dichotomous; values are discrete
Redundancy among cues	High redundancy	Low redundancy
Decomposition of task	Low	High
Degree of certainty in task	Low certainty	High certainty
Relation between cues and criterion	Linear	Nonlinear
Weighting of cues in environmental model	Equal	Unequal
Availability of organizing principle	Unavailable	Available
Display of cues	Simultaneous display	Sequential display
Time period	Brief	Long

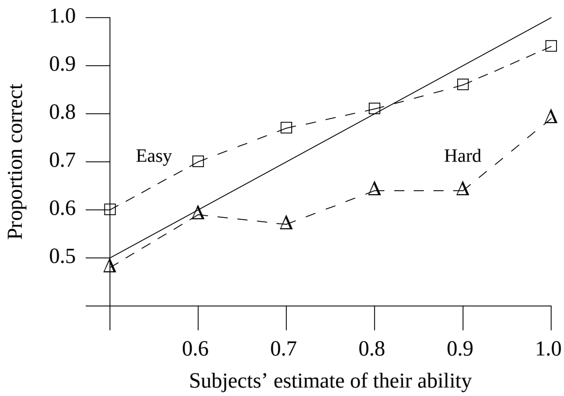


Figure 0.18: Subjects' estimate of their ability (bottom scale) to correctly answer a question and actual performance in answering on the left scale. The responses of a person with perfect self-knowledge is given by the solid line. Adapted from Lichtenstein.[111]

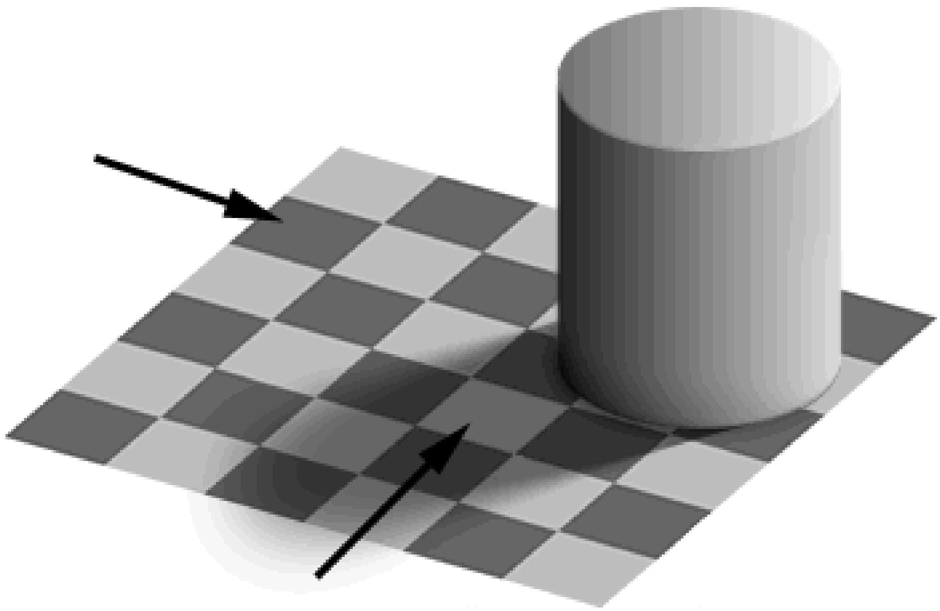


Figure 0.19: Checker shadow (by Edward Adelson). Which of the two squares indicated by the arrows is the brighter one (following inverted text gives answer)? Both squares reflect the same amount of light (this can be verified by covering all of squares except the two indicated), but the human visual system assigns a relative brightness that is consistent with the checker pattern.

Table 0.13: Cognitive anomalies. Adapted from McFadden.[119]

Effect	Description
CONTEXT	
Anchoring	Judgments are influenced by quantitative cues contained in the statement of the decision task
Context	Prior choices and available options in the decision task influence perception and motivation
Framing	Selection between mathematically equivalent solutions to a problem depends on how their outcome is framed.
Prominence	The format in which a decision task is stated influences the weight given to different aspects
REFERENCE POINT	
Risk asymmetry	Subjects show risk-aversion for gains, risk-preference for losses, and weigh losses more heavily

Figure 0.20: The Thatcher illusion. With permission from Thompson.[165] The facial images look very similar when viewed in one orientation and very different when viewed in another (turn page upside down).

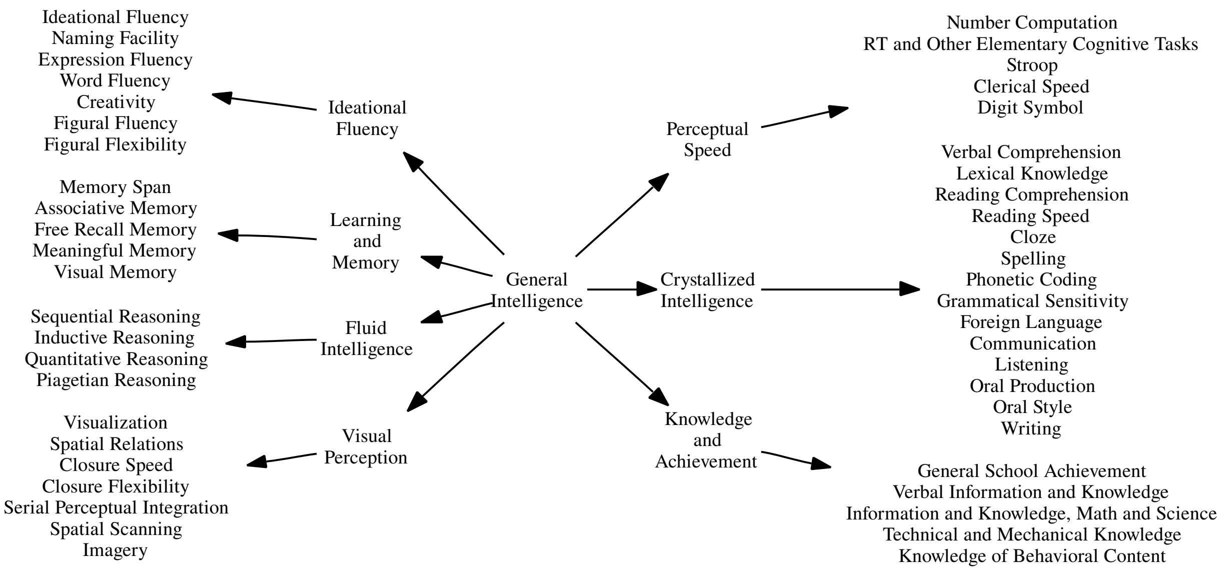


Figure 0.21: A list of and structure of ability constructs. Adapted from Ackerman.[1]

Table 0.14: Words with either one or more than one syllable (and thus varying in the length of time taken to speak).

List 1	List 2	List 3	List 4	List 5
one	cat	card	harm	add
bank	lift	list	bank	mark
sit	able	inch	view	bar
kind	held	act	fact	few
look	mean	what	time	sum
ability	basically	encountered	laboratory	commitment
particular	yesterday	government	acceptable	minority
mathematical	department	financial	university	battery
categorize	satisfied	absolutely	meaningful	opportunity
inadequate	beautiful	together	carefully	accidental

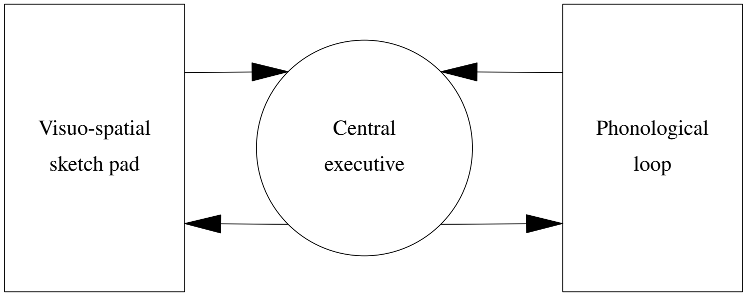


Figure 0.22: Model of working memory. Adapted from Baddeley.[10]

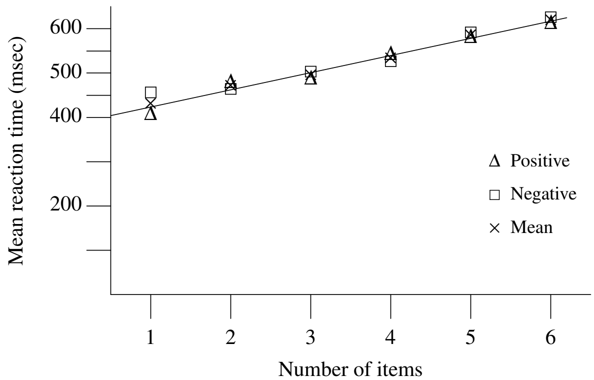


Figure 0.23: Judgment time (in milliseconds) as a function of the number of digits held in memory. Adapted from Sternberg.[157]

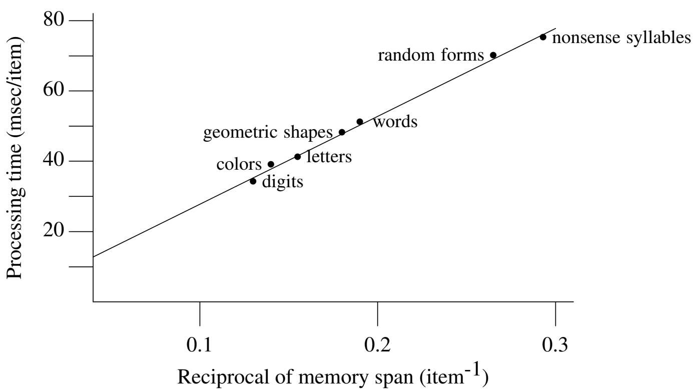


Figure 0.24: Judgment time (msec per item) as a function of the number of different items held in memory. Adapted from Cavanagh[33]

Table 0.15: Proactive inhibition. The third row indicates learning performance; the fifth row indicates recall performance, relative to that of the control. Based on Anderson.[6]

Subject 1	Subject 2	Subject 3
Learn A⇒B Learn A⇒D	Learn C⇒D Learn A⇒B	Rest Learn A⇒D
Worse	Better	
Test A⇒D	Test A⇒D	Test A⇒D
Worse	Worse	

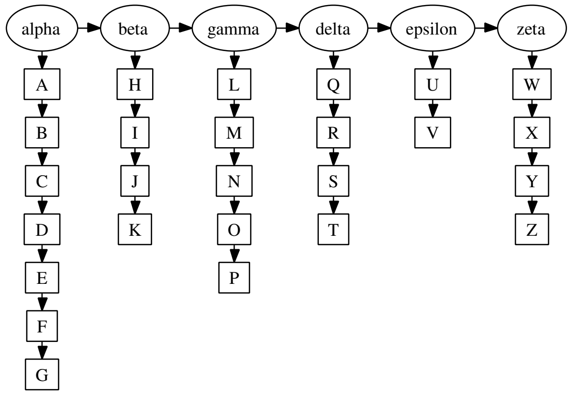


Figure 0.25: Semantic memory representation of alphabetic letters (the Greek names assigned to nodes by Klahr are used by the search algorithm and are not actually held in memory). Readers may recognize the structure of a nursery rhyme in the letter sequences. Derived from Klahr.[96]

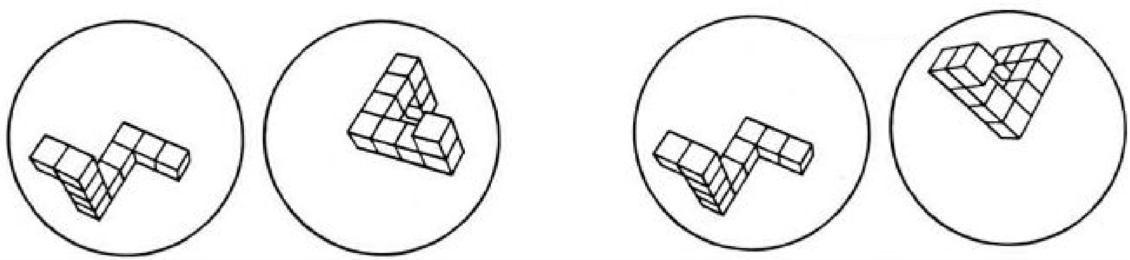


Figure 0.26: One of the two pairs are rotated copies of each other.

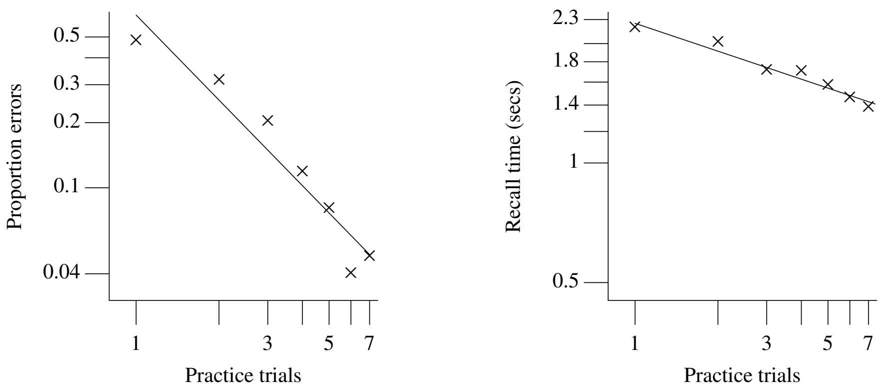


Figure 0.27: Proportion of errors (left) and time to recall (right) for recall of paired associate words (log scale). Based on Anderson.[5]

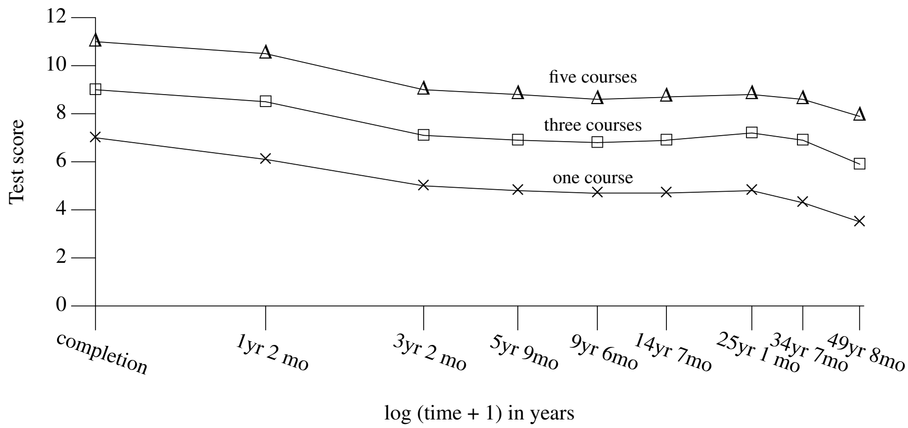


Figure 0.28: Effect of level of training on the retention of recognition of English–Spanish vocabulary. Adapted from Bahrick.[11]

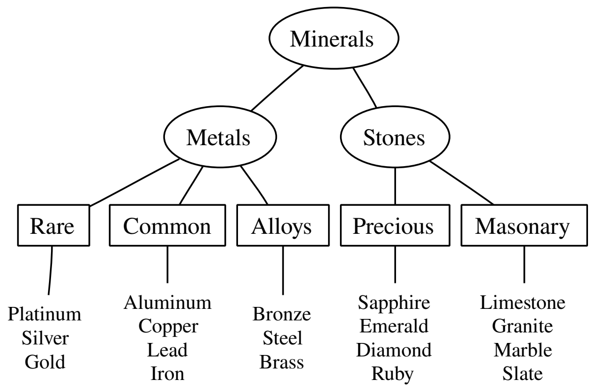


Figure 0.29: Words organized according to their properties— the *minerals* conceptual hierarchy. Adapted from Bower, Clark, Lesgold, and Winzenz.[20]

Table 0.16: Retroactive inhibition. The fourth row indicates subject performance relative to that of the control. Based on Anderson.[6]

Subject 1	Subject 2	Subject 3
Learn A⇒B	Learn A⇒B	Learn A⇒B
Learn A⇒D	Learn C⇒D	Rest
Test A⇒B	Test A⇒B	Test A⇒B
Much worse	Worse	

Table 0.17: Main failure modes for skill-based performance. Adapted from Reason.[143]

Inattention	Over Attention
Double-capture slips	Omissions
Omissions following interruptions	Repetitions
Reduced intentionality	Reversals
Perceptual confusions	
Interference errors	

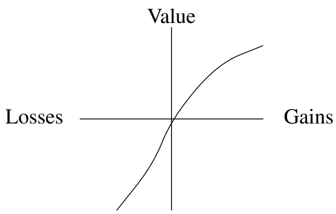


Figure 0.30: Relationship between subjective value to gains and to losses. Adapted from Kahneman.[92]

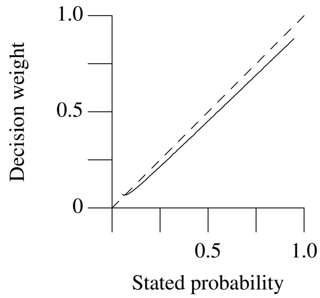


Figure 0.31: Possible relationship between subjective and objective probability. Adapted from Kahneman.[92]

Table 0.18: Main failure modes for knowledge-based performance. Adapted from Reason.[143]

Knowledge-based Failure Modes
Selectivity
Workspace limitations
Out of, sight out of mind
Confirmation bias
Overconfidence
Biased reviewing
Illusory correlation
Halo effects
Problems with causality
Problems with complexity
Problems with delayed feed-back
Insufficient consideration of processes in time
Difficulties with exponential developments
Thinking in causal series not causal nets (unaware of side-effects of action)
Thematic vagabonding (flitting from issue to issue)
Encysting (lingering in small detail over topics)

Table 0.19: Main failure modes for rule-base performance. Adapted from Reason.[143]

Misapplication of Good Rules	Application of Bad Rules
First exceptions	Encoding deficiencies
Countersigns and nosigns	Action deficiencies
Information overload	Wrong rules
Rule strength	Inelegant rules
General rules	Inadvisable rules
Redundancy	
Rigidity	

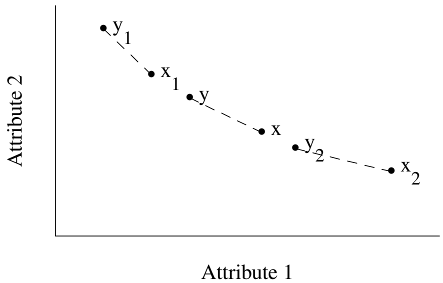


Figure 0.32: Text of background trade-off. Adapted from Tversky.[169]

Table 0.20: Percentage of each alternative selected by subject groups S_1 and S_2. Adapted from Tversky.[169]

Warranty	Price	S_1	S_2
X_1	$85	12%	
Y_1	$91	88%	
X_2	$25		84%
Y_2	$49		16%
X	$60	57%	33%
Y	$75	43%	67%

Table 0.21: Percentage of subjects willing to exchange what they had been given for an equivalently priced item. Adapted from Knetsch.[98]

Group	Yes	No
Give up mug to obtain candy	89%	11%
Give up candy to obtain mug	90%	10%

Table 0.22: Percentage of subjects giving each answer. Correct answers are starred. Adapted from Kahneman.[91]

Choice	Less than 6	More than 6
The page investigator	20.8%*	16.3%
The line investigator	31.3%	42.9%*
About the same (i.e., within 5% of each other)	47.9%	40.8%

Usage

1 Introduction

Usage 1
Usage introduction

This subsection provides some background on the information appearing in the Usage subsections of this book. The purpose of this usage information is two-fold:

1. To give readers a feel for the common developer usage of C language constructs. Part of the process of becoming an experienced developers involves learning about what is common and what is uncommon. However, individual experiences can be specific to one application domain, or company cultures.

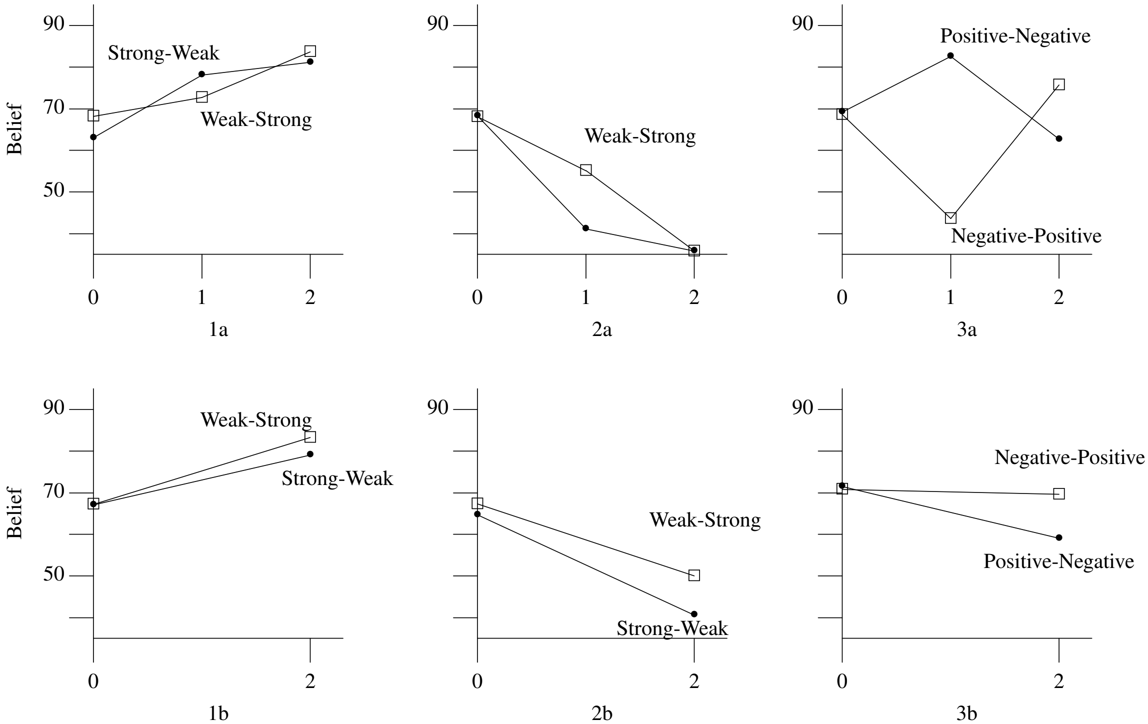


Figure 0.33: Subjects belief response curves for positive weak–strong, negative weak–strong, and positive–negative evidence; (a) Step-by-Step, (b) End-of-Sequence. Adapted from Hogarth.[83]

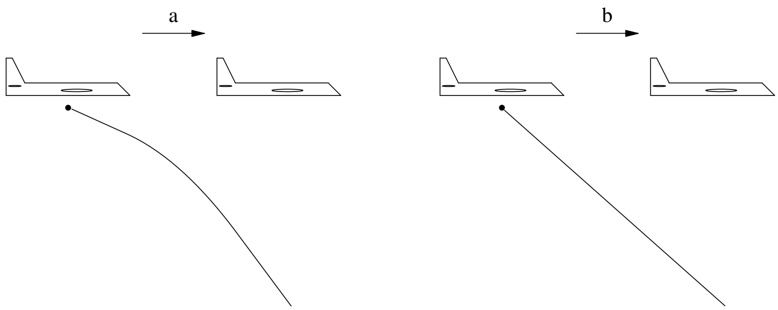


Figure 0.34: Two proposed trajectories of a ball dropped from a moving airplane. Based on McCloskey.[118]

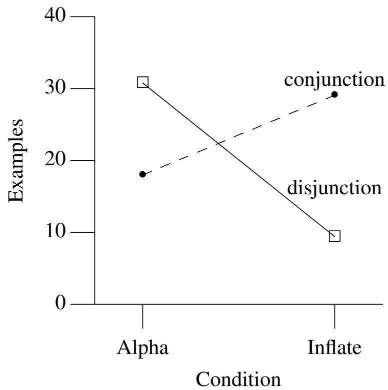


Figure 0.35: Number of examples needed before *alpha* or *inflate* condition correctly predicted in six successive pictures. Adapted from Pazzani[136]

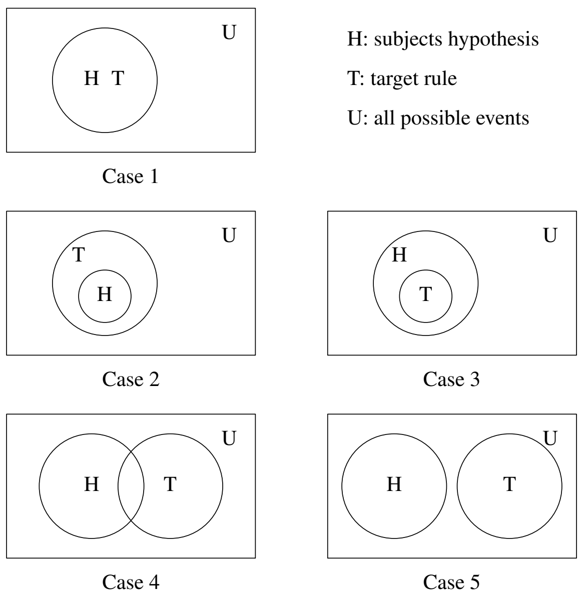


Figure 0.36: Possible relationships between hypothesis and rule. Adapted from Klayman.[97]

2. To provide frequency-of-occurrence information that could be used as one of the inputs to cost/benefit decisions (i.e., should a guideline recommendation be made rather than what recommendation might be made). This is something of a chicken-and-egg situation in that knowing what measurements to make requires having potential guideline recommendations in mind, and the results of measurements may suggest guideline recommendations (i.e., some construct occurs frequently). guideline recommendations selecting

Almost all published measurements on C usage are an adjunct to a discussion of some translator optimization technique. They are intended to show that the optimization, which is the subject of the paper, is worthwhile because some constructs occurs sufficiently often for an optimization to make worthwhile savings, or that some special cases can be ignored because they rarely occur. These kinds of measurements are usually discussed in the *Common implementation* subsections. One common difference between the measurements in Common Implementation subsections and those in Usage subsections is that the former are often dynamic (instruction counts from executing programs), while the latter are often static (counts based on some representation of the source code).

There have been a few studies whose aim has been to provide a picture of the kinds of C constructs that commonly occur (e.g., preprocessor usage,[63] embedded systems[61]). These studies are quoted in the relevant C sentences. There have also been a number of studies of source code usage for other algorithmic languages, Assembler,[48] Fortran,[99] PL/1,[59] Cobol[4,38,88] (measurements involving nonalgorithmic languages have very different interests[32,41]). These are of interest in studying cross-language usage, but they are not discussed in this book. In some cases a small number of machine code instruction sequences (which might be called idioms) have been found to account for a significant percentage of the instructions executed during program execution.[153]

The intent here is to provide a broad brush picture. On the whole, single numbers are given for the number of occurrences of a construct. In most cases there is no break down by percentage of functions, source files, programs, application domain, or developer. There is variation across all of these (e.g., application domain and individual developer). Whenever this variation might be significant, additional information is given. Those interested in more detailed information might like to make their own measurements. coding guidelines applications coding guidelines coding style

Many of the coding guideline recommendations made in this book apply to the visible source code as seen by the developer. For these cases any usage measurements also apply to the visible source code. The effects of any macro replacement, conditional inclusion, or `#include`d header are ignored. Each usage subsection specifies what the quoted numbers apply to (usually either visible source, or the tokens processed during translation phase 7). macro replacement

In practice many applications do not execute in isolation; there is usually some form of operating system that is running concurrently with it. The design of processor instruction sets often takes task-switching and other program execution management tasks into account. In practice the dynamic profile of instructions executed by a processor reflects this mix of usage,[17] as does the contents of its cache.[117]

1.1 Characteristics of the source code

source code characteristics

All source code may appear to look the same to the casual observer. An experienced developer will be aware of recurring patterns; source can be said to have a style. Several influences can affect the characteristics of source code, including the following: coding guidelines coding style

- *Use of extensions to the C language and differences, for prestandard C, from the standard (often known as K&R C).* Some extensions eventually may be incorporated into a revised version of the standard; for instance, **long long** was added in C99. Some extensions are specific to the processor on which the translated program is to execute. common implementations language specification
- *The application domain.* For instance, scientific and engineering applications tend to make extensive use of arrays and spend a large amount of their time in loops processing information held in these arrays; screen based interactive applications often contain many calls to GUI library functions and can spend more time in these functions than the developer's code; data-mining applications can spend a significant amount of time searching large data structures.

- *How the application is structured.* Some applications consist of a single, monolithic, program, while others are built from a collection of smaller programs sharing data with one another. These kinds of organization affect how types and objects are defined and used.

application evolution

- *The extent to which the source has evolved over time.* Developers often adopt the low-risk strategy of making the minimal number of changes to a program when modifying it. Often this means that functions and sequences of related statements tend to grow much larger than would be the case if they had been written from scratch, because no restructuring is performed.

macro object-like 1931

- *Individual or development group stylistic usage.* These differences can include the use of large or small functions, the use of enumeration constants or object-like macros, the use of the smallest integer type required rather than always using **`int`**, and so forth.

1.2 What source code to measure?

This book is aimed at a particular audience and the source code they are likely to be actively working on. This audience will be working on C source that has been written by more than one developer, has existed for a year or more, and is expected to continue to be worked on over the coming years.

benchmarks

The benchmarks used in various application areas were written with design aims that differ from those of this book. For instance, the design aim behind the choice of programs in the SPEC CPU benchmark suite was to measure processor, memory hierarchy, and translator performance. Many of these programs were written by individuals, are relatively short, and have not changed much over time.

Although there is a plentiful supply of C source code publicly available (an estimated 20.3 million C source files on the Web[18]), this source is nonrepresentative in a number of ways, including:

- The source has had many of the original defects removed from it. The ideal time to make these measurements is while the source is being actively developed.
- Software for embedded systems is often so specialized (in the sense of being tied to custom hardware), or commercially valuable, that significant amounts of it are not usually made publicly available.

Nevertheless, a collection of programs was selected for measurement, and the results are included in this book (see Table 0.23). The programs used for this set of measurements have reached the stage that somebody has decided that they are worth releasing. This means that some defects in the source, prior to the release, will not be available to be included in these usage figures.

Table 0.23: Programs whose source code (i.e., the `.c` and `.h` files) was used as the input to measurement tools (operating on either the visible or translated forms), whose output was used to generate this book's usage figures and tables.

Name	Application Domain	Version
gcc	C compiler	2.95
idsoftware	Games programs, e.g., Doom	
linux	Operating system	2.4.20
mozilla	Web browser	1.0
openafs	File system	1.2.2a
openMotif	Window manager	2.2.2
postgresql	Database system	6.5.3

Table 0.24: Source files excluded from the Usage measurements.

Files	Reason for Exclusion
gcc-2.95/libio/tests/tfformat.c	a list of approximately 4,000 floating constants
gcc-2.95/libio/tests/tiformat.c	a list of approximately 5,000 hexadecimal constants

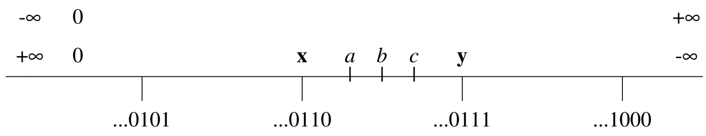


Figure 64.1: Some exactly representable values and three values (a, b, and c) that are not exactly representable.

Table 0.25: Character sequences used to denote those operators and punctuators that perform more than one role in the syntax.

Symbol	Meaning	Symbol	Meaning
++v	prefix ++	--v	prefix --
v++	postfix ++	v--	postfix --
-v	unary minus	+v	unary plus
*v	indirection operator	*p	star in pointer declaration
&v	address-of		
:b	colon in bitfield declaration	?:	colon in ternary operator

42 **implementation-defined behavior** implementation-defined behavior

unspecified behavior where each implementation documents how the choice is made

Usage

Annex J.3 lists 97 implementation-defined behaviors.

46 **undefined behavior** undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

Usage

Annex J.2 lists 190 undefined behaviors.

49 **unspecified behavior** unspecified behavior

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

Usage

Annex J.1 lists 50 unspecified behaviors.

63 **constraint** constraint

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

Usage

There are 134 instances of the word *shall* in a Constraints clause (out of 588 in the complete standard). This places a lower bound on the number of constraints that can be violated (some uses of *shall* describe more than one kind of construct).

64 **correctly rounded result** correctly rounded result

representation in the result format that is nearest in value, subject to the ~~effective~~current rounding mode, to what the result would be given unlimited range and precision

82 In this International Standard, “shall” is to be interpreted as a requirement on an implementation or on a program; shall

Usage

The word *shall* occurs 537 times (excluding occurrences of *shall not*) in the C Standard.

---

conversely, “shall not” is to be interpreted as a prohibition. 83

Usage

The phrase *shall not* occurs 51 times (this includes two occurrences in footnotes) in the C Standard.

---

shall outside constraint

If a “shall” or “shall not” requirement that appears outside of a constraint is violated, the behavior is undefined. 84

Usage

The word *shall* appears 454 times outside of a Constraint clause; however, annex J.2 only lists 190 undefined behaviors. The other uses of the word *shall* apply to requirements on implementations, not programs.

---

implementation two forms

The two forms of *conforming implementation* are hosted and freestanding. 92

---

footnote 3

3) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard. 98

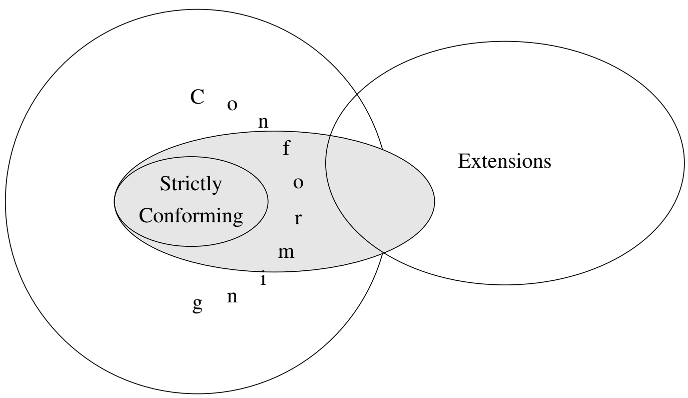


Figure 92.1: A conforming implementation (gray area) correctly handles all strictly conforming programs, may successfully translate and execute some of the possible conforming programs, and may include some of the possible extensions.

Table 98.1: Number of developer declared identifiers (the contents of any header was only counted once) whose spelling (the notation *[a-z]* denotes a regular expression, i.e., a character between *a* and *z*) is reserved for use by the implementation or future revisions of the C Standard. Based on the translated form of this book's benchmark programs.

Reserved spelling	Occurrences
Identifier, starting with __, declared to have any form	3,071
Identifier, starting with *_[A-Z]*, declared to have any form	10,255
Identifier, starting with *wcs[a-z]*, declared to have any form	1
Identifier, with external linkage, defined in C99	12
File scope identifier or tag	6,832
File scope identifier	2
Macro name reserved when appropriate header is `#included`	6
Possible macro covered identifier	144
Macro name starting with *E[A-Z]*	339
Macro name starting with *SIG[A-Z]*	2
Identifier, starting with *is[a-z]*, with external linkage (possibly macro covered)	47
Identifier, starting with *mem[a-z]*, with external linkage (possibly macro covered)	108
Identifier, starting with *str[a-z]*, with external linkage (possibly macro covered)	904
Identifier, starting with *to[a-z]*, with external linkage (possibly macro covered)	338
Identifier, starting with *is[a-z]*, with external linkage	33
Identifier, starting with *mem[a-z]*, with external linkage	7
Identifier, starting with *str[a-z]*, with external linkage	28
Identifier, starting with *to[a-z]*, with external linkage	62

107 A C program need not all be translated at the same time. program not translated at same time

Usage

A study by Linton and Quong[113] used an instrumented `make` program to investigate the characteristics of programs (written in a variety of languages, including C) built over a six-month period at Stanford University. The results (see Figure 107.1) showed that approximately 40% of programs consisted of three or fewer translation units.

116 1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. translation phase 1

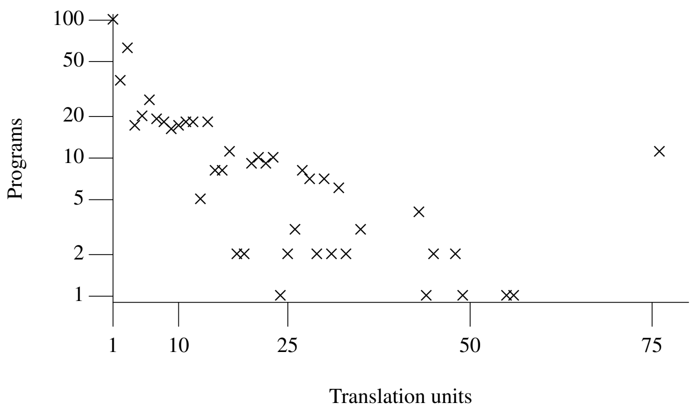


Figure 107.1: Number of programs built from a given number of translation units. Adapted from Linton.[113]

Table 116.1: Total number of characters and new-lines in the visible form of the .c and .h files.

	.c files	.h files
total characters	192,165,594	64,429,463
total new-lines	6,976,266	1,811,790
non-comment characters	144,568,262	43,485,916
non-comment new-lines	6,113,075	1,491,192

trigraph sequences phase 1

Trigraph sequences are replaced by corresponding single-character internal representations. 117

Usage

The visible form of the .c files contain 8 trigraphs (.h 0).

translation phase 2
physical source line
logical source line

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. 118

Usage

In the visible form of the .c files 0.21% (.h 4.7%) of all physical lines are spliced. Of these line splices 33% (.h 7.8%) did not occur within preprocessing directives (mostly in string literals).

translation phase 3

3. The source file is decomposed into preprocessing tokens[6)] and sequences of white-space characters (including comments). 124

Usage

preprocessing tokens 777
white space separation

The visible form of the .c files contain 30,901,028 (.h 8,338,968) preprocessing tokens (new-line not included); 531,677 (.h 248,877) /* */ comments, and 52,531 (.h 27,393) // comments.

Usage information on white space is given elsewhere.

translation phase 6

6. Adjacent string literal tokens are concatenated. 135

Usage

In the visible form of the .c files 4.9% (.h 15.6%) of string literals are concatenated.

program image

All such translator output is collected into a program image which contains information needed for execution in its execution environment. 141

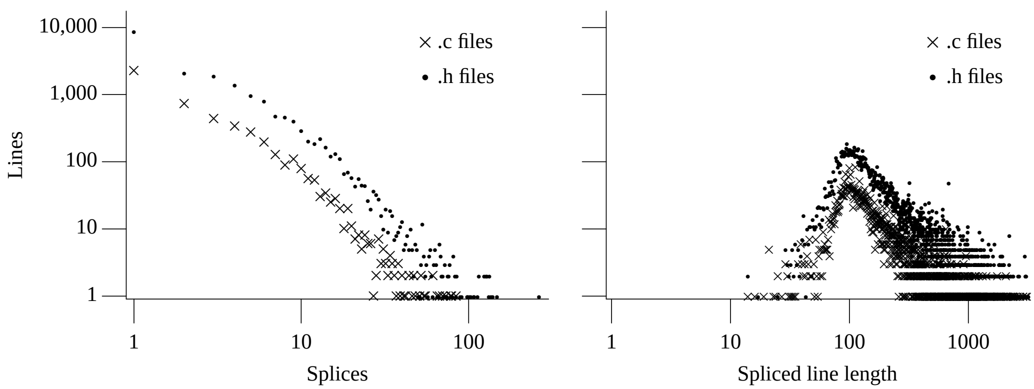


Figure 118.1: Number of physical lines spliced together to form one logical line and the number of logical lines, of a given length, after splicing. Based on the visible form of the .c and .h files.

Table 141.1: *Total* is the number of code pages in the application; *Touched* the number of code pages touched during startup; *Utilization* the average fraction of functions used during startup in each code page. Adapted from Lee.[106]

Application	Total	Touched (%)	% Utilization
acrobat	404	246 (60)	28
netscape	388	388 (100)	26
photoshop	594	479 (80)	28
powerpoint	766	164 (21)	32
word	743	300 (40)	47

164 It shall be defined with a return type of **`int`** and with no parameters:

```
int main(void) { /* ... */ }
```

Usage

There was not a sufficiently large number of instances of `main` in the `.c` files to provide a reliable measure of the different ways this function is declared.

190 An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object). expression need not evaluate part of

221 Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin alphabet basic source character set basic execution character set

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

the 26 *lowercase letters* of the Latin alphabet

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

the 10 decimal *digits*

```
0 1 2 3 4 5 6 7 8 9
```

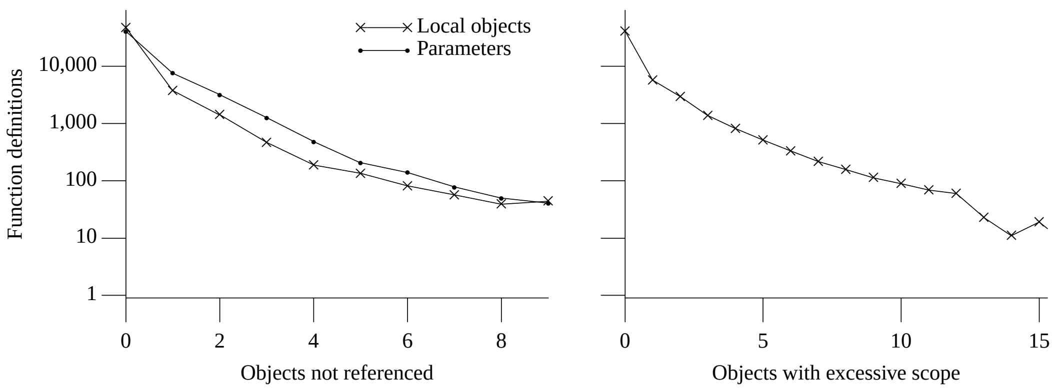


Figure 190.1: Number of parameters or locally defined objects that are not referenced within a function definition (left graph); number of objects declared in a scope that is wider than that needed for any of the references to them within a function definition (right graph). Based on the translated form of this book's benchmark programs.

the following 29 graphic characters

```
!  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed.

Table 221.1: Occurrence of characters as a percentage of all characters and as a percentage of all noncomment characters (i.e., outside of comments). Based on the visible form of the .c files. For a comparison of letter usage in English language and identifiers see Figure 792.16.

Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment
0	0.000	0.000	**sp**	15.083	13.927	**@**	0.009	0.002	**‘**	0.004	0.002
1	0.000	0.000	**!**	0.102	0.127	**A**	0.592	0.642	**a**	3.132	2.830
2	0.000	0.000	**"**	0.376	0.471	**B**	0.258	0.287	**b**	0.846	0.812
3	0.000	0.000	**#**	0.175	0.219	**C**	0.607	0.663	**c**	2.168	2.178
4	0.000	0.000	**$**	0.005	0.003	**D**	0.461	0.523	**d**	2.184	2.176
5	0.000	0.000	**%**	0.105	0.135	**E**	0.869	1.012	**e**	5.642	4.981
6	0.000	0.000	**&**	0.237	0.311	**F**	0.333	0.355	**f**	1.666	1.725
7	0.000	0.000	**’**	0.101	0.080	**G**	0.243	0.263	**g**	0.923	0.906
8	0.000	0.000	**(**	1.372	1.751	**H**	0.146	0.155	**h**	1.145	0.777
\t	3.350	4.116	**)**	1.373	1.751	**I**	0.619	0.643	**i**	3.639	3.469
\n	3.630	4.229	*****	1.769	0.769	**J**	0.024	0.026	**j**	0.074	0.077
11	0.000	0.000	**+**	0.182	0.233	**K**	0.098	0.116	**k**	0.464	0.481
12	0.003	0.004	**,**	1.565	1.914	**L**	0.528	0.609	**l**	2.033	1.915
\r	0.001	0.001	**-**	1.176	0.831	**M**	0.333	0.366	**m**	1.245	1.229
14	0.000	0.000	**.**	0.512	0.387	**N**	0.557	0.610	**n**	3.225	2.989
15	0.000	0.000	/	0.718	0.519	**O**	0.467	0.517	**o**	2.784	2.328
16	0.000	0.000	**0**	1.465	1.694	**P**	0.460	0.508	**p**	1.505	1.551
17	0.000	0.000	**1**	0.502	0.551	**Q**	0.033	0.037	**q**	0.121	0.135
18	0.000	0.000	**2**	0.352	0.408	**R**	0.652	0.729	**r**	3.405	3.254
19	0.000	0.000	**3**	0.227	0.262	**S**	0.691	0.758	**s**	3.166	2.961
20	0.000	0.000	**4**	0.177	0.203	**T**	0.686	0.740	**t**	4.566	4.200
21	0.000	0.000	**5**	0.149	0.171	**U**	0.315	0.349	**u**	1.575	1.510
22	0.000	0.000	**6**	0.176	0.209	**V**	0.128	0.149	**v**	0.662	0.682
23	0.000	0.000	**7**	0.131	0.144	**W**	0.131	0.135	**w**	0.494	0.385
24	0.000	0.000	**8**	0.184	0.207	**X**	0.213	0.254	**x**	0.870	1.002
25	0.000	0.000	**9**	0.128	0.122	**Y**	0.091	0.094	**y**	0.515	0.435
26	0.000	0.000	**:**	0.192	0.176	**Z**	0.027	0.033	**z**	0.125	0.135
27	0.000	0.000	**;**	1.276	1.670	**[**	0.163	0.210	**{**	0.303	0.401
28	0.000	0.000	**<**	0.118	0.147	\	0.097	0.126	|	0.098	0.124
29	0.000	0.000	**=**	1.039	1.042	**]**	0.163	0.210	**}**	0.303	0.401
30	0.000	0.000	**>**	0.587	0.762	**^**	0.003	0.002	**~**	0.009	0.012
31	0.000	0.000	**?**	0.022	0.019	**_**	2.550	3.238	**127**	0.000	0.000

Table 221.2: Relative frequency (most common to least common, with parenthesis used to bracket extremely rare letters) of letter usage in various human languages (the English ranking is based on the British National Corpus). Based on Kelk.[94]

Language	Letters
English	etaoinsrhldcumfpgwybvkxjqz
French	esaitnrulodcmpévqfbghjàxèyêzâçîùôûïkëw
Norwegian	erntsilakodgmvfupbhøjyåæcwzx(q)
Swedish	eantrsildomkgväfhupåöbcyjxwzéq
Icelandic	anriestuðlgmkfhvoáþídjóbyæúöpéýcxwzq
Hungarian	eatlnskomzrigáéydbvhjőfupöócűíúüxw(q)

---

trigraph sequences mappings

233

```
??= #   ??) ]   ??! |
??( [   ??' ^   ??< }
??/ \   ??< {   ??- ~
```

Usage

There are insufficient trigraphs in the visible form of the `.c` files to enable any meaningful analysis of the usage of different trigraphs to be made.

---

trigraph sequences no other

234 No other trigraph sequences exist.

Usage

The visible form of the `.c` files contained 593 (`.h` 10) instances of two question marks (i.e., `??`) in string literals that were not followed by a character that would have created a trigraph sequence.

---

multibyte character state-dependent encoding shift state

243 — A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence.

Table 243.1: Commonly seen ISO 2022 Control Characters. The alternative values for SS2 and SS3 are only available for 8-bit codes.

Name	Acronym	Code Value	Meaning
Escape	ESC	0x1b	Escape
Shift-In	SI	0x0f	Shift to the G0 set
Shift-Out	SO	0x0e	Shift to the G1 set
Locking-Shift 2	LS2	ESC 0x6e	Shift to the G2 set
Locking-Shift 3	LS3	ESC 0x6f	Shift to the G3 set
Single-Shift 2	SS2	ESC 0x4e, or 0x8e	Next character only is in G2
Single-Shift 3	SS3	ESC 0x4f, or 0x8f	Next character only is in G3

Table 243.2: An implementation where G1 is ISO 8859–1, and G2 is ISO 8891–7 (Greek).

Encoded values	0x62	0x63	0x64	0x0e	0xe6	0x1b	0x6e	0xe1	0xe2	0xe3	0x0f
Control character				SO		LS2					SI
Graphic character	a	b	c		æ			α	β	γ	

Table 243.3: ESC codes for some of the character sets used in Japanese.

Character Set	Byte Encoding	Visible Ascii Representation
JIS C 6226–1978	1B 24 40	<ESC> $ @
JIS X 0208–1983	1B 24 42	<ESC> $ B
JIS X 0208–1990	1B 26 40 1B 24 42	<ESC> & @ <ESC> $ B
JIS X 0212–1990	1B 24 28 44	<ESC> $ (D
JIS-Roman	1B 28 4A	<ESC> (J
Ascii	1B 28 42	<ESC> (B
Half width Katakana	1B 28 49	<ESC> (I

Table 243.4: A JIS encoding of the character sequence 171202193250 (“kana and kanji”).

Encoded values	0x1b	0x24	0x42	0x242b	0x244a	0x3441	0x3b7a	0x1b	0x28	0x4a
Control character	<ESC>	$	B					<ESC>	(	J
Graphic character				か	な	漢	字			
Ascii characters				$+	$J	4A	;z			

limit
block nesting

— 127 nesting levels of blocks 277

— 63 nesting levels of conditional inclusion 278

limit
type complexity

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration 279

parenthesized expression nesting levels

— 63 nesting levels of parenthesized expressions within a full expression 281

internal identifier
significant characters

— 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character) 282

Usage

Very few identifiers approach the C99 translation limit (see Figure 792.7).

external identifier
significant characters

— 31 significant initial characters in an external identifier (each universal character name specifying a short 283

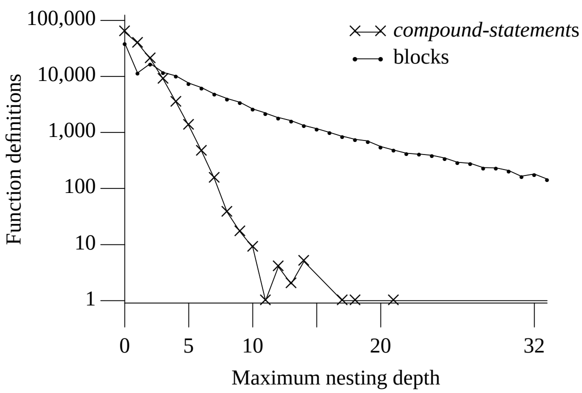


Figure 277.1: Number of functions containing blocks and *compound-statements* nested to the given maximum nesting level. Based on the visible form of the .c files.

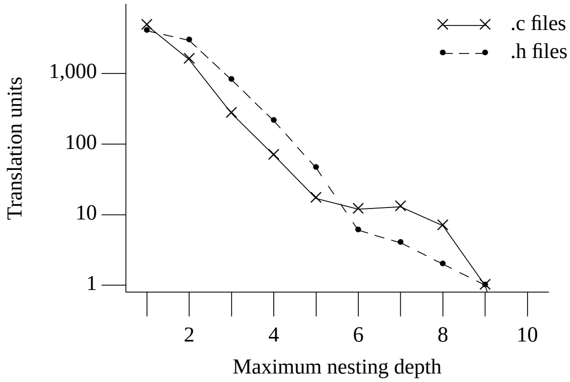


Figure 278.1: Number of translation units containing conditional inclusion directives nested to the given maximum nesting level. Based on the visible form of the `.c` and `.h` files.

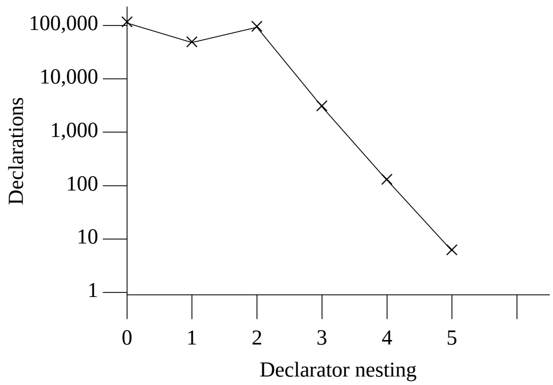


Figure 279.1: Number of full declarators containing a given number of modifiers. Based on the translated form of this book's benchmark programs.

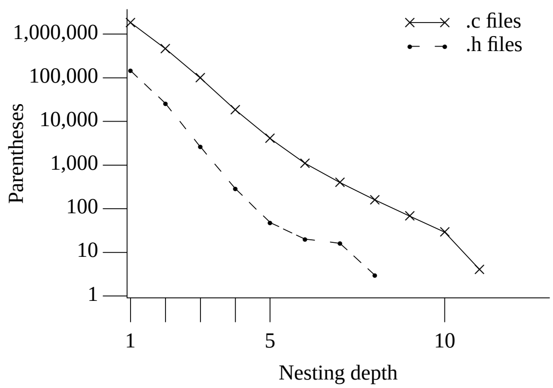


Figure 281.1: Nesting of all occurrences of parentheses. Based on the visible form of the `.c` and `.h` files.

identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[14]

limit external identifiers

— 4095 external identifiers in one translation unit 285

Usage

External declaration usage information is given elsewhere (see Figure 1810.1).

Table 285.1: Number of identifiers with external linkage (total 487), and total number of identifiers (total 810), implementations are required to declare in the standard headers.

Header	External Identifiers	Total Identifiers	Header	External Identifiers	Total Identifiers
<assert.h>	1	2	<signal.h>	2	12
<complex.h>	66	71	<stdarg.h>	3	5
<ctype.h>	15	15	<stdbool.h>	0	4
<errno.h>	1	4	<stddef.h>	0	5
<fenv.h>	11	24	<stdint.h>	0	38
<float.h>	0	31	<stdio.h>	49	65
<inttypes.h>	6	62	<stdlib.h>	36	37
<iso646.h>	0	11	<string.h>	22	24
<limits.h>	0	19	<tgmath.h>	0	60
<locale.h>	2	10	<time.h>	9	15
<math.h>	184	203	<wchar.h>	59	68
<setjmp.h>	2	3	<wctype.h>	18	22

identifiers number in block scope

— 511 identifiers with block scope declared in one block 286

Usage

The 53,630 function definitions in the translated form of this book's benchmark programs contained: definitions of 76 structure, union or enumeration types that included a tag; 6 **typedef** definitions; and definitions of 70 enumeration constants.

limit macro definitions

— 4095 macro identifiers simultaneously defined in one preprocessing translation unit 287

limit parameters in definition

— 127 parameters in one function definition 288

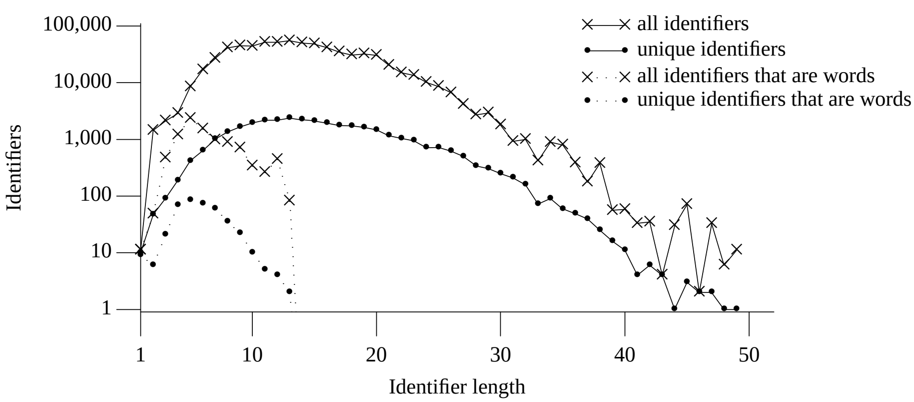


Figure 283.1: Number of identifiers, with external linkage, having a given length. Based on the translated form of this book's benchmark programs. Information on the length of all identifiers in the visible source is given elsewhere (see Figure 792.7).

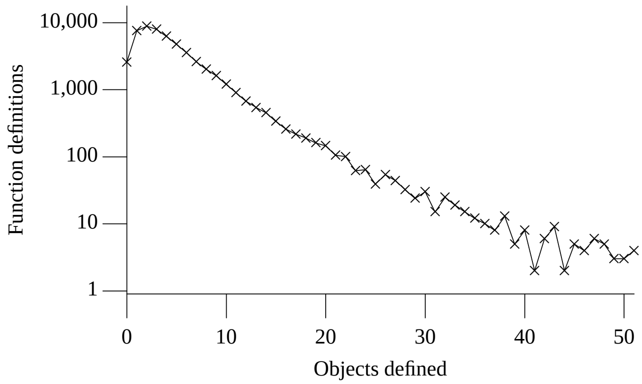


Figure 286.1: Number of function definitions containing a given number of definitions of identifiers as objects. Based on the translated form of this book's benchmark programs.

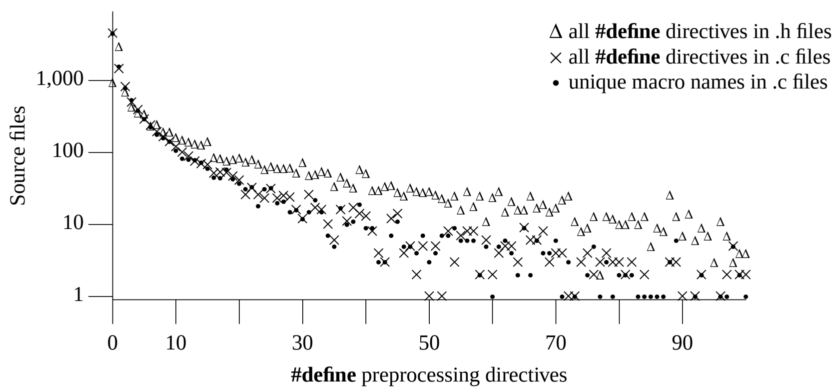


Figure 287.1: Number of source files containing a given number of identifiers defined as macro names in **`#define`** preprocessing directives. Unique macro name counts an identifier once, irrespective of the number of **`#define`** directives it appears in. Based on the visible form of the `.c` and `.h` files.

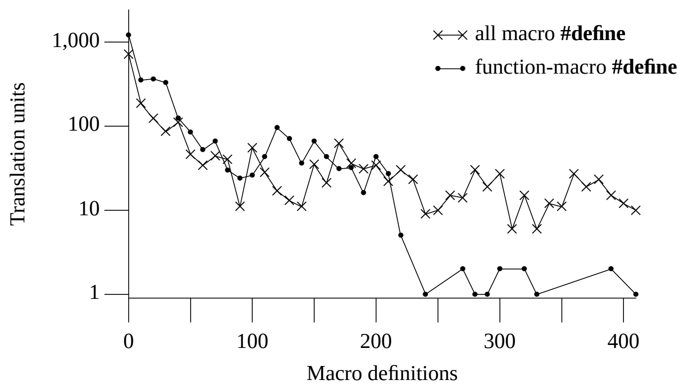


Figure 287.2: Number of translation units containing a given number of evaluations of **`#define`** preprocessing directives, excluding the contents of system headers, during translation of this book's benchmark programs (there were a total of 1,432,735 macros defined, of which 313,620 were function-like macros).

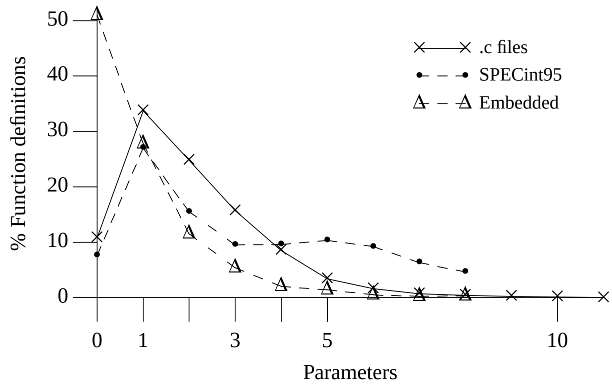


Figure 288.1: Percentage of function definitions appearing in the source of embedded applications (5,597 function definitions), the SPECint95 benchmark (2,713 function definitions), and the translated form of this book's benchmark programs (53,719 function definitions) declared to have a given number of parameters. The embedded and SPECint95 figures are from Engblom.[61]

function call number of arguments

— 127 arguments in one function call 289

limit macro parameters

— 127 parameters in one macro definition 290

limit arguments in macro invocation

— 127 arguments in one macro invocation 291

limit characters on line

— 4095 characters in a logical source line 292

limit string literal

— 4095 characters in a character string literal or wide string literal (after concatenation) 293

limit minimum object size

— 65535 bytes in an object (in a hosted environment only) 294

limit #include nesting

— 15 nesting levels for **`#include`**d files 295

limit case labels

— 1023 **`case`** labels for a **`switch`** statement (excluding those for any nested **`switch`** statements) 296

limit members in struct/union

— 1023 members in a single structure or union 297

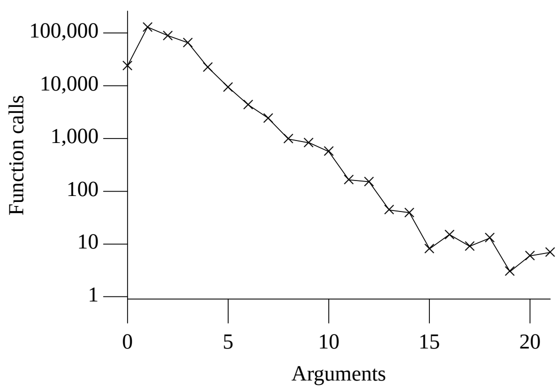


Figure 289.1: Number of function calls containing a given number of arguments. Based on the translated form of this book's benchmark programs.

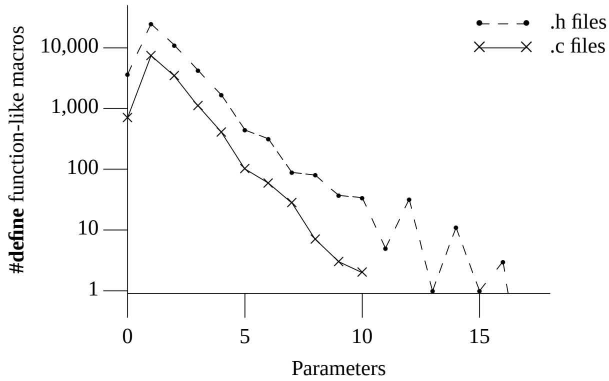


Figure 290.1: Number of function-like macro definitions having a given number of parameter declarations. Based on the visible form of the `.c` and `.h` files.

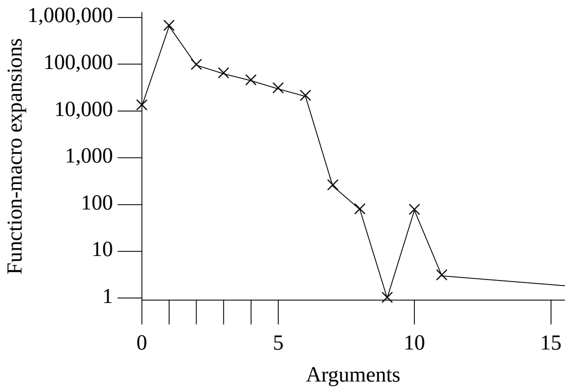


Figure 291.1: Number of function-like macro expansions containing a given number of arguments, excluding expansions that occurred while processing system headers, during translation of this book's benchmark programs.

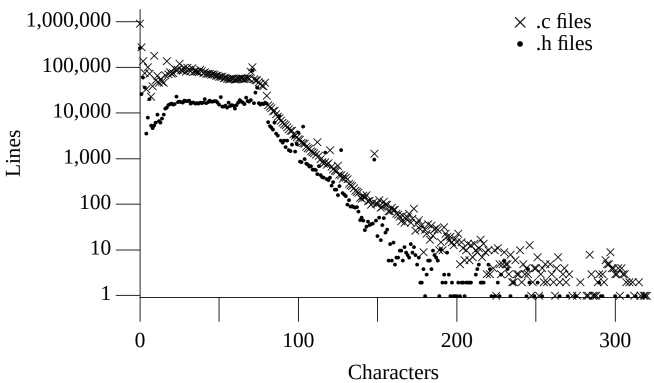


Figure 292.1: Number of physical lines containing a given number of characters. Based on the visible form of the `.c` and `.h` files.

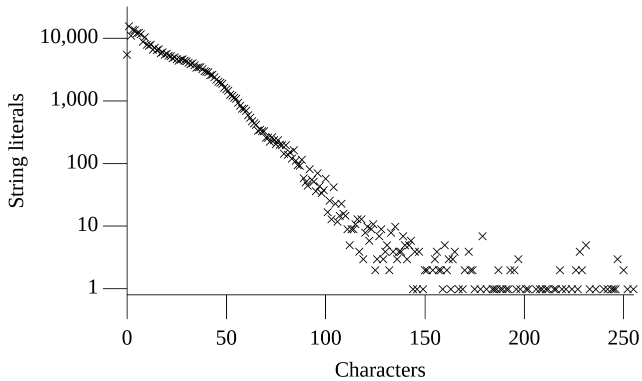


Figure 293.1: Number of character string literals containing a given number of characters (i.e., their length). Based on the visible form of the `.c` files.

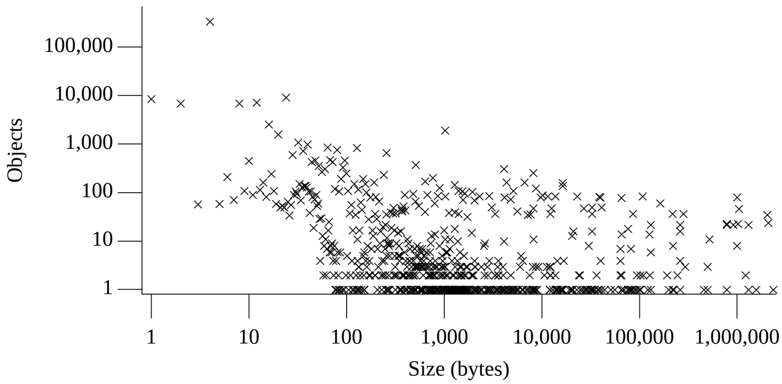


Figure 294.1: Number of objects requiring the specified amount of storage. Based on the translated form of this book's benchmark programs, using integer types whose sizes were: `sizeof(short) == 2`, `sizeof(int) == 4`, and `sizeof(long) == 4`; and alignment requirements that were a multiple of a types size.

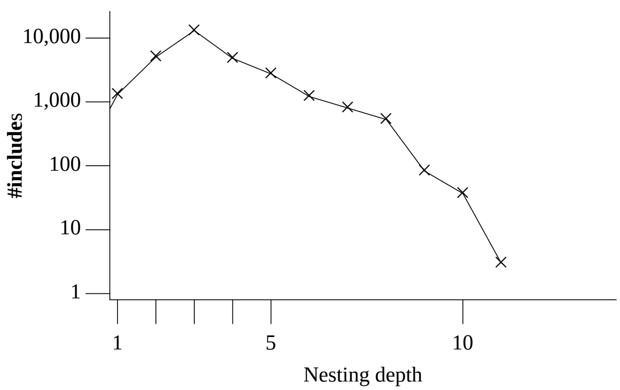


Figure 295.1: Number of **`#include`** preprocessor directives, that contain the quote-delimited form of header name (occurrences of the `< >` delimited form were not counted), having a given nesting depth. Based on the translated form of this book's benchmark programs.

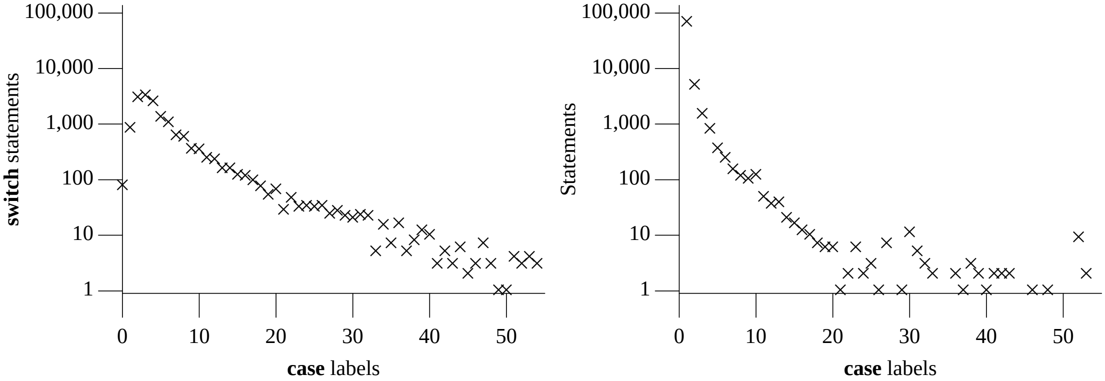


Figure 296.1: Number of **`switch`** statements containing the given number of **`case`** labels (left) and number of individual statements labeled by a given number of **`case`** labels (right). Based on the visible form of the `.c` files. Note that counts do not include occurrences of the **`default`** label.

Usage

Measurements of classes,[173] in large Java programs, have found that the number of members follows the same pattern as that in C (see Figure 297.1).

---

298 — 1023 enumeration constants in a single enumeration limit enumeration constants

---

299 — 63 levels of nested structure or union definitions in a single struct-declaration-list limit struct/union nesting

---

303 The values given below shall be replaced by constant expressions suitable for use in **`#if`** preprocessing directives. integer types sizes

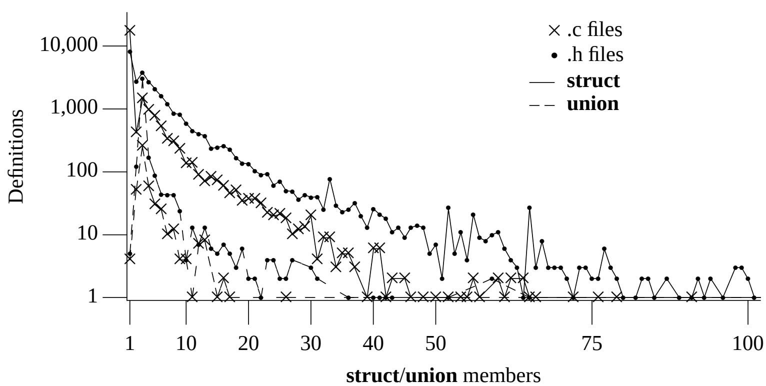


Figure 297.1: Number of structure and union type definitions containing the given number of members (members in any nested definitions are not included in the count of members of the outer definition). Based on the visible form of the `.c` and `.h` files.

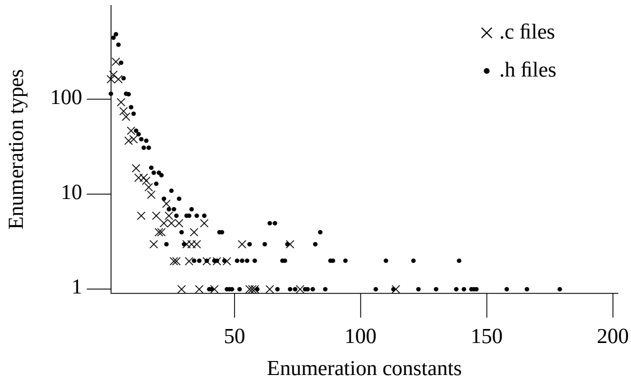


Figure 298.1: Number of enumeration types containing a given number of enumeration constants. Based on the visible form of the .c and .h files (also see Figure 1439.1).

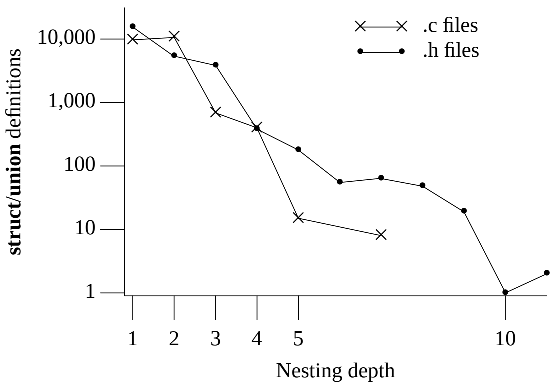


Figure 299.1: Number of structure and union type definitions containing the given number nested members that are textually structure and union type definitions (i.e., definitions using **{ }** not typedef names). Based on the visible form of the .c and .h files.

Table 303.1: Number of identifiers defined as macros in **<limits.h>** (see Table 770.3 for information on the number of identifiers appearing in the source) appearing in the visible form of the .c and .h files.

Name	.c file	.h file	Name	.c file	.h file	Name	.c file	.h file
LONG_MAX	47	28	CHAR_MAX	15	8	CHAR_BIT	36	3
INT_MAX	106	17	INT_MIN	17	7	SCHAR_MIN	12	2
UINT_MAX	30	14	UCHAR_MAX	16	5	LLONG_MAX	0	1
SHRT_MAX	20	13	CHAR_MIN	9	5	ULLONG_MAX	0	0
SHRT_MIN	19	12	SCHAR_MAX	13	4	LLONG_MIN	0	0
USHRT_MAX	12	11	MB_LEN_MAX	15	4			
ULONG_MAX	85	10	LONG_MIN	23	3			

floating types characteristics

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[16)] 330

Table 330.1: Range of representable floating-point values for the Unisys e-@ction Application Development Solutions Compiling System.

Type	Bits	Decimal Range
float	36	1.4693680E-39 . . . 1.7014118E+38
double	72	2.7813423E-309 . . . 8.9884657E+307
long double	72	2.7813423E-309 . . . 8.9884657E+307

Table 330.2: Area of triangle, using Heron's formula, calculated using different rounding directions.

	Correct	Rounding Down	Rounding Up
x	100.01	100.01	100.01
y	99.995	99.995	99.995
z	0.025	0.025	0.025
$(x+(y+z))/2$	100.015	100.01	100.02
Area	1.000025	0.0000	1.5813

Usage

Many of the following identifiers were referenced from one program, `enquire.c`, whose job was to deduce the characteristics of a host's floating-point support.

Table 330.3: Number of identifiers defined as macros in **`<float.h>`** (see Table 770.3 for information on the number of identifiers appearing in the source) appearing in the visible form of the `.c` and `.h` files.

Name	.c file	.h file	Name	.c file	.h file	Name	.c file	.h file
DBL_MIN	9	21	FLT_MAX	5	15	FLT_ROUNDS	18	14
DBL_MAX	20	19	FLT_DIG	5	15	FLT_RADIX	20	14
DBL_DIG	41	17	LDBL_MIN_EXP	4	14	FLT_MIN_EXP	4	14
FLT_EPSILON	4	16	LDBL_MIN	4	14	FLT_MIN_10_EXP	4	14
DBL_MIN_EXP	4	16	LDBL_MIN_10_EXP	4	14	FLT_MAX_EXP	4	14
DBL_MIN_10_EXP	4	16	LDBL_MAX_EXP	4	14	FLT_MAX_10_EXP	4	14
DBL_MAX_EXP	27	16	LDBL_MAX	4	14	FLT_MANT_DIG	8	14
DBL_MAX_10_EXP	14	16	LDBL_MAX_10_EXP	4	14	FLT_EVAL_METHOD	0	0
DBL_MANT_DIG	14	16	LDBL_MANT_DIG	4	14	DECIMAL_DIG	0	0
DBL_EPSILON	4	16	LDBL_EPSILON	4	14			
FLT_MIN	5	15	LDBL_DIG	4	14			

334 e exponent (an integer between a minimum e_{min} and a maximum e_{max}) exponent

Usage

The range of exponent values that can occur within programs may depend on the application domain. For instance, astronomy programs may contain ranges of very large values and subatomic particle programs contain ranges of very small values. A study of software for automotive control systems[45] showed (see Table 334.1) a relatively small range of exponents, close to zero.

Table 334.1: Dynamic distribution of decimal exponents, as a percentage, for operands of various floating point operations. Adapted from Connors, Yamada, and Hwu[45] (thanks to Connors for supplying the raw data).

Exponent	Compare	Add	Multiply	Divide	Exponent	Compare	Add	Multiply	Divide
0	15.60	11.4	6.7	3.0					
-1	2.5	2.5	1.9	0.0	1	10.80	9.3	1.6	1.0
-2	0.7	1.2	0.6	1.0	2	5.20	2.6	1.3	3.0
-3	0.1	0.0	0.7	0.0	3	8.50	4.3	0.7	0.0
-4	0.0	0.1	0.2	1.0	4	0.50	0.0	0.5	0.0
-5	0.0	0.0	0.5	0.0					
-6	0.0	0.6	1.4	0.0					

floating types can represent

In addition to normalized floating-point numbers ($f_1 > 0$ if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers ($x \neq 0$, $e = e_{min}$, $f_1 = 0$) and unnormalized floating-point numbers ($x \neq 0$, $e > e_{min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. 338

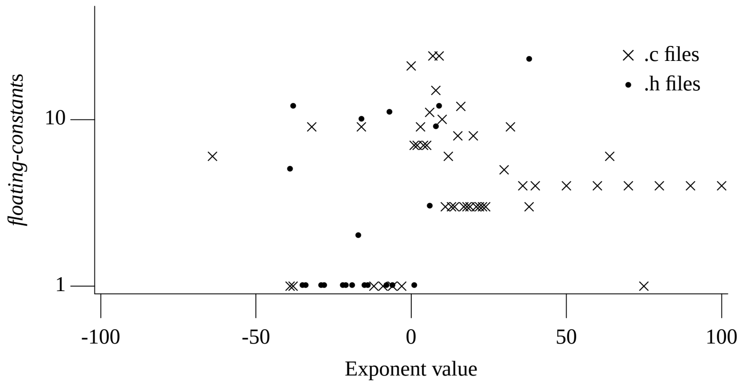


Figure 334.1: Number of *floating-constants* (that included an *exponent-part*) having a given exponent value. Based on the visible form of the .c and .h files.

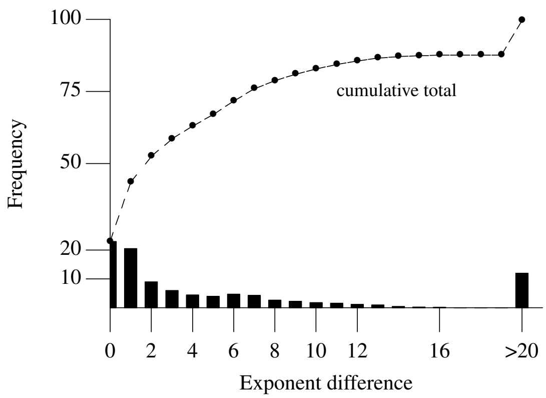


Figure 334.2: Difference in the value of the exponents (in powers of 2) of the two operands of floating-point addition and subtraction operations, obtained by executing the SPECfp92 benchmarks. Adapted from Oberman.[131]

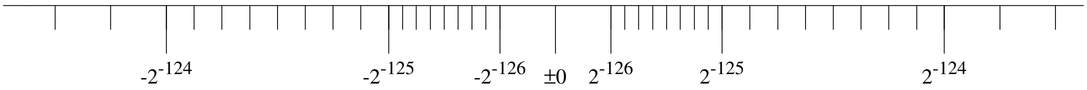


Figure 338.1: Range of normalized numbers about zero, including subnormal numbers.

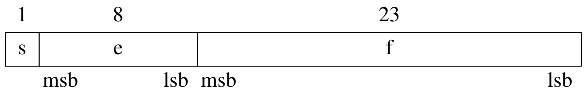


Figure 338.2: Single-precision IEC 60559 format.

Table 338.1: Format Parameters of IEC 60559 representation. All widths measured in bits. Intel's *extended-precision* format is a conforming IEC 60559 format derived from that standards *extended double-precision* format.

Parameter	Single	Single Extended	Double	Double Extended	Intel x86 Extended
Precision, p, (apparent mantissa width)	24	32	53	64	64
Actual mantissa width	23	31	52	63	64
Mantissa's MS-Bit	hidden bit	unspecified	hidden bit	unspecified	explicit bit
Decimal digits of precision, $p/\log_2(10)$	7.22	9.63	15.95	19.26	19.26
E_{max}	+127	+1023	+1023	+16383	+16383
E_{min}	-126	-1022	-1022	-16382	-16382
Exponent bias	+127	unspecified	+1023	unspecified	+16383
Exponent width	8	11	11	15	15
Sign width	1	1	1	1	1
Format width (9) + (8) + (4)	32	43	64	79	80
Maximum value, $2^{E_{max}+1}$	3.4028E+38	1.7976E+308	1.7976E+308	1.1897E+4932	1.1897E+4932
Minimum value, $2^{E_{min}}$	1.1754E-38	2.2250E-308	2.2250E-308	3.3621E-4932	3.3621E-4932
Denormalized minimum value, $2^{E_{min}-(4)}$	1.4012E-45	1.0361E-317	4.9406E-324	3.6451E-4951	1.8225E-4951

Table 338.2: List of some results of operations on infinities and NaNs. Also see: "Expression transformations" in annex F.8.2 of the C Standard.

$Operation \Longrightarrow Result$	$Operation \Longrightarrow Result$
$x/(+\infty) \Longrightarrow +0$	$x/(+0) \Longrightarrow +\infty$
$x/(-\infty) \Longrightarrow -0$	$x/(-0) \Longrightarrow -\infty$
$(+\infty) + x \Longrightarrow +\infty$	$x + NaN \Longrightarrow NaN$
$(+\infty) \times x \Longrightarrow +\infty$	$\infty \times 0 \Longrightarrow NaN$
$(+\infty)/x \Longrightarrow +\infty$	$0/0 \Longrightarrow NaN$
$(+\infty) - (+\infty) \Longrightarrow NaN$	$NaN - NaN \Longrightarrow NaN$

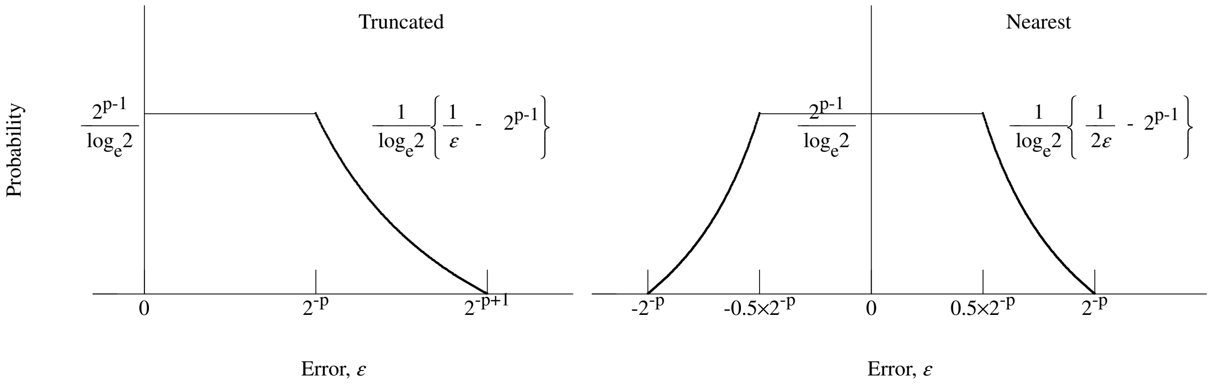


Figure 346.1: Probability of a floating-point operation having a given error (ϵ) for two kinds of rounding modes (truncated and to-nearest); p is the number of digits in the significand. Adapted from Tsao.[101]

Table 338.3: Example of gradual underflow. * Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

Variable or Operation	Value	Biased Exponent	Comment
A0	1.100 1100 1100 1100 1100 1101 × 2^{-125}	2	
A1 = A0 / 2	1.100 1100 1100 1100 1100 1101 × 2^{-126}	1	
A2 = A1 / 2	0.110 0110 0110 0110 0110 0110 × 2^{-126}	0	Inexact*
A3 = A2 / 2	0.011 0011 0011 0011 0011 0011 × 2^{-126}	0	Exact result
A4 = A3 / 2	0.001 1001 1001 1001 1001 1010 × 2^{-126}	0	Inexact*
.			
.			
.			
A23 = A22 / 2	0.000 0000 0000 0000 0000 0011 × 2^{-126}	0	Exact result
A24 = A23 / 2	0.000 0000 0000 0000 0000 0010 × 2^{-126}	0	Inexact*
A25 = A24 / 2	0.000 0000 0000 0000 0000 0001 × 2^{-126}	0	Exact result
A26 = A25 / 2	0.0	0	Inexact*

floating-point operations accuracy

The accuracy of the floating-point operations (+, -, *, /) and of the library functions in **<math.h>** and **<complex.h>** that return floating-point results is implementation-defined , as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library routine in **<stdio.h>**, **<stdlib.h>** and **<wchar.h>**. 346

Usage

In theory it is possible to measure the accuracy required/expected by an application. However, it is not possible to do this automatically — it requires detailed manual analysis. Consequently, there are no usage figures for this sentence (because no such analyses have been carried out by your author for any of the programs in the measurement set).

FLT_ROUNDS

The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**:18) 352

-1 indeterminable
0 toward zero
1 to nearest
2 toward positive infinity
3 toward negative infinity

All other values for **FLT_ROUNDS** characterize implementation-defined rounding behavior.

Table 352.1: Effect of rounding mode (FLT_ROUNDS taking on values 0, 1, 2, or 3) on the result of a single precision value (given in the left column).

	0	1	2	3
1.00000007	1.0	1.00000012	1.00000012	1.0
1.00000003	1.0	1.0	1.00000012	1.0
-1.00000003	-1.0	-1.0	-1.0	-1.00000012
-1.00000007	-1.0	-1.00000012	-1.0	-1.00000012

DECIMAL_DIG macro

368 — number of decimal digits, n, such that any floating-point number in the widest supported floating type with p_{max} radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p_{max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p_{max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

DECIMAL_DIG **10**

---

389 A summary of the language syntax is given in annex A.

---

identifier denote different entities

397 The same identifier can denote different entities at different points in the program.

---

file scope

407 If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit.

---

block scope terminates

408 If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block.

---

outer scope identifier hidden

413 the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

Usage

In the translated form of this book's benchmark programs there were 1,945 identifier definitions (out of 270,394 identifiers defined in block scope) where an identifier declared in an inner scope hid an identifier declared in an outer block scope.

---

linkage

420 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[21)]

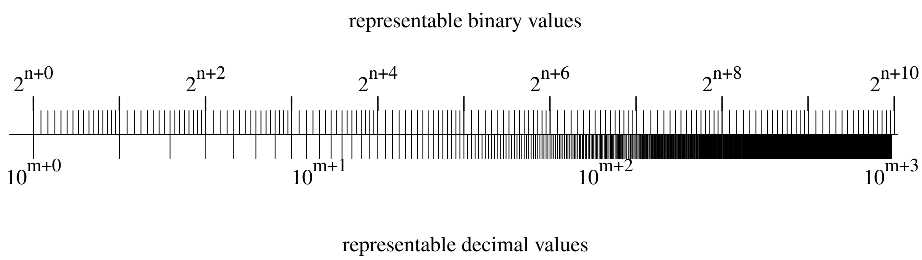


Figure 368.1: Representable powers of 10 and powers of 2 on the real line.

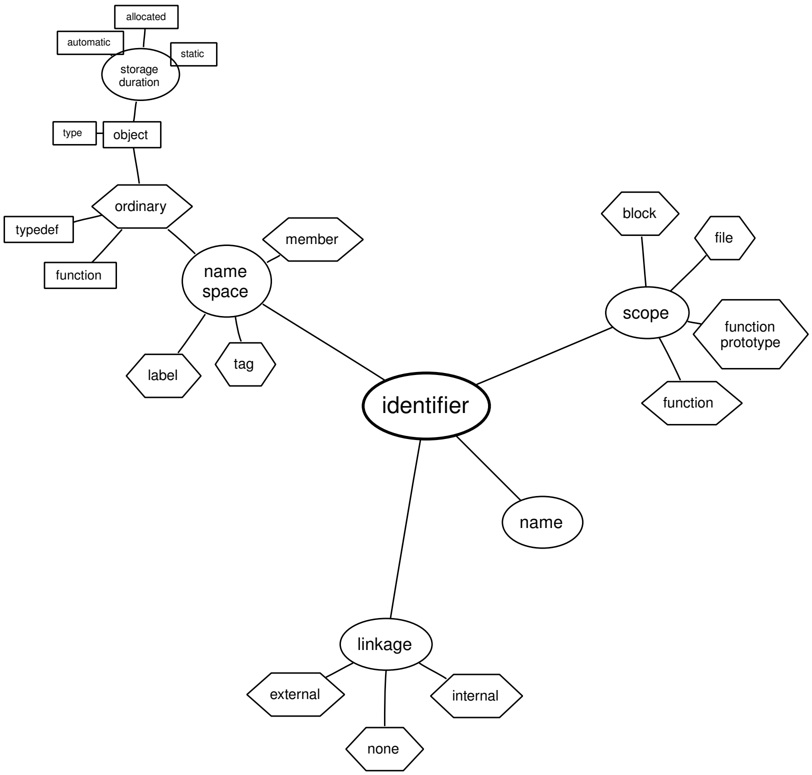


Figure 389.1: Attributes a C language identifier can have.

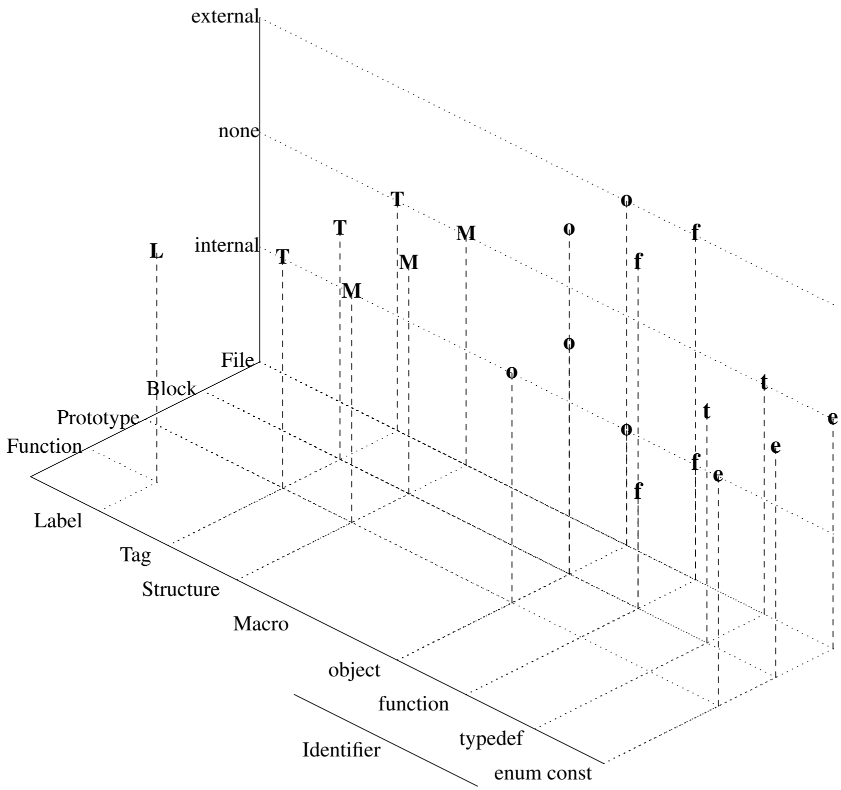


Figure 397.1: All combinations of linkage, scope, and name space that all possible kinds of identifiers, supported by C, can have. **M** refers to the members of a structure or union type. There is a separate name space for macro names and they have *no linkage*, but their scope has no formally specified name.

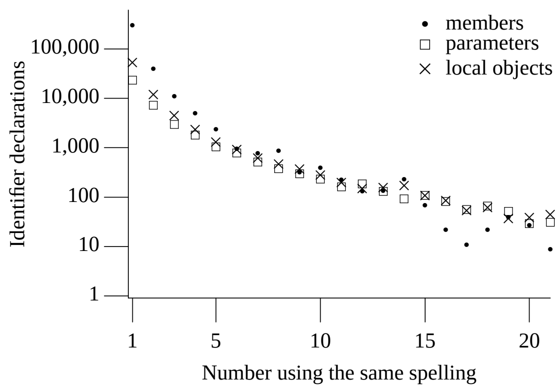


Figure 397.2: Number of declarations of an identifier with the same spelling in the same translation unit. Based on the translated form of this book's benchmark programs. Note that members of the same type are likely to be counted more than once (i.e., they are counted in every translation unit that declares them), while parameters and objects declared within function definitions are likely to be only counted once.

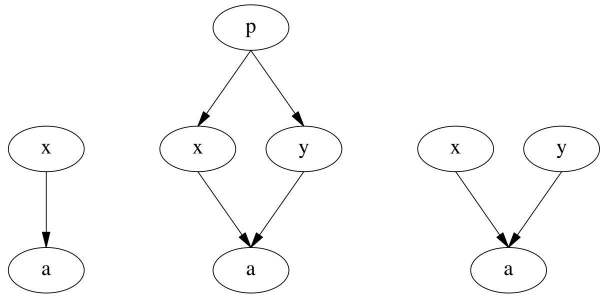


Figure 407.1: Some of the ways in which a function can be called— a single call from one other function; called from two or more functions, which in turn are all called by a single function; and called from two or more functions whose nearest shared calling function is not immediately above them.

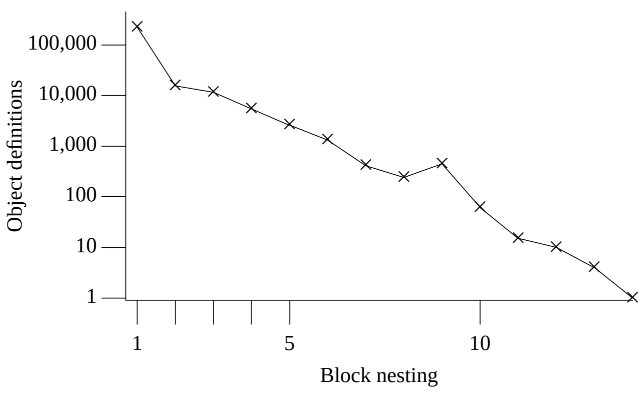


Figure 408.1: Number of object declarations appearing at various block nesting levels (level 1 is the outermost block). Based on the translated form of this book's benchmark programs.

Table 420.1: Comparison of identifier linkage models

Model	File 1	File 2
common	`extern int I; int main() { I = 1; second(); }`	`extern int I; void second() { third( I ); }`
Relaxed Ref/Def	`int I; int main() { I = 1; second(); }`	`int I; void second() { third( I ); }`
Strict Ref/Def	`int I; int main() { I = 1; second(); }`	`extern int I; void second() { third( I ); }`
Initializer	`int I = 0; int main() { I = 1; second(); }`	`int I; void second() { third( I ); }`

---

421 There are three kinds of linkage: external, internal, and none. linkage kinds of

---

422 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. object external linkage denotes same function external linkage denotes same

Usage

A study of 29 Open Source programs by Srivastava, Hicks, Foster and Jenkins[154] found 1,161 identifiers with external linkage, referenced in more than one translation unit, that were not declared in a header, and 809 instances where a header containing the declaration of a referenced identifier was not **#include**d (i.e., the source file contained a textual external declaration of the identifier).

---

436 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined. linkage both internal/external

Usage

The translated form of this book's benchmark programs contained 27 instances of identifiers declared, within the same translation unit, with both internal and external linkage.

---

438 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. name space

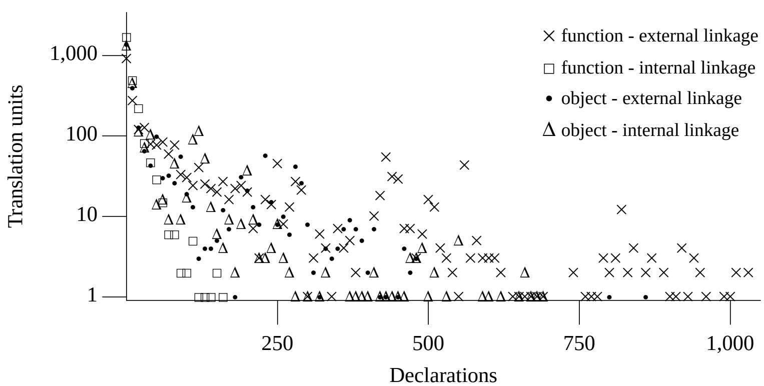


Figure 421.1: Number of translation units containing a given number of objects and functions declared with internal and external linkage (excluding declarations in system headers). Based on the translated form of this book's benchmark programs.

Table 438.1: Identifiers appearing immediately to the right of the given token as a percentage of all instances of the given token. An identifier appearing to the left of a **:** could be a label or a **case** label. However, C syntax is designed to be parsed from left to right and the presence, or absence, of a **case** keyword indicates the entity denoted by an identifier. Based on the visible form of the `.c` files.

Token	`.c` file	`.h` file	Token	`.c` file	`.h` file
goto identifier	99.9	100.0	**struct** identifier	99.0	88.4
#define identifier	99.9	100.0	**union** identifier	65.5	75.8
. identifier	100.0	99.8	**enum** identifier	86.6	53.6
-> identifier	100.0	95.5	**case** identifier	71.3	47.2

Thus, there are separate *name spaces* for various categories of identifiers, as follows: 439

Table 439.1: Occurrence of various kinds of declarations of identifiers as a percentage of all identifiers declared in all the given contexts. Based on the translated form of this book's benchmark programs.

Declaration Context	%	Declaration Context	%
block scope objects	23.7	file scope objects	4.4
macro definitions	19.3	macro parameters	4.3
function parameters	16.8	enumeration constants	2.1
struct/union members	9.6	**typedef** names	1.2
function declarations	8.6	tag names	1.0
function definitions	8.1	label names	0.9

member namespace

each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the . or -> operator); 443

Table 443.1: Number of matches found when comparing between pairs of members contained in different Pascal records that were defined with the same type name. Adapted from Anquetil and Lethbridge.[7]

	Member Types the Same	Member Types Different	Total
Member names the same	73 (94.8%)	4 (5.2%)	77
Member names different	52 (11 %)	421 (89 %)	473

Table 443.2: Number of matches found when comparing between pairs of members contained in different Pascal records (that were defined with the any type name). Adapted from Anquetil and Lethbridge.[7]

	Member Types the Same	Member Types Different	Total
Member names the same	7,709 (33.7%)	15,174 (66.3%)	22,883
Member names different	158,828 (0.2%)	66,652,062 (99.8%)	66,710,890

There are three storage durations: static, automatic, and allocated. 449

Usage

In the translated form of this book's benchmark programs 37% of defined objects had static storage duration and 63% had automatic storage duration (objects with allocated storage duration were not included in this count).

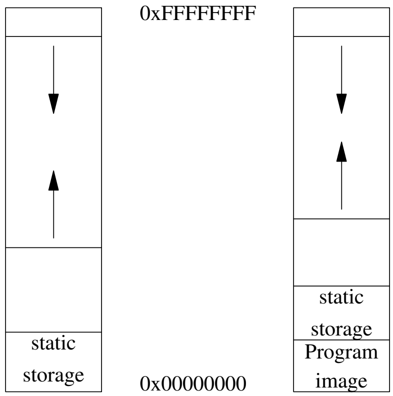


Figure 449.1: The location of the stack invariably depends on the effect of a processor's pop/push instructions (if they exist). The heap usually goes at the opposite end of available storage. The program image may, or may not, exist in the same address space.

Table 449.1: Total number of objects allocated (in thousands), the total amount of storage they occupy (in thousands of bytes), their average size (in bytes) and the high water mark of these values (also in thousands). Adapted from Detlefs, Dosser and Zorn.[54]

Program	Total Objects	Total Bytes	Average Size	Maximum Objects	Maximum Bytes
sis	63,395	15,797,173	249.2	48.5	1,932.2
perl	1,604	34,089	21.3	2.3	116.4
xfig	25	1,852	72.7	19.8	1,129.3
ghost	924	89,782	97.2	26.5	2,129.0
make	23	539	23.0	10.4	208.1
espresso	1,675	107,062	63.9	4.4	280.1
ptc	103	2,386	23.2	102.7	2,385.8
gawk	1,704	67,559	39.6	1.6	41.0
cfrac	522	8001	15.3	1.5	21.4

---

455 An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **`static`** has *static storage duration*. static storage duration

Usage

In the visible form of the `.c` files approximately 5% of occurrences of the keyword **`static`** occurred in block scope.

---

462 If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; initialization performed every time declaration reached

Usage

Usage information on initializers is given elsewhere.

1652 object value indeterminate

---

464 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.27) VLA lifetime starts/ends

---

479 If any other character is stored in a **`char`** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

Usage

In the visible form of the `.c` files 2.1% (`.h` 2.9%) of characters in character constants are not in the basic execution character set (assuming the Ascii character set representation is used for escape sequences).

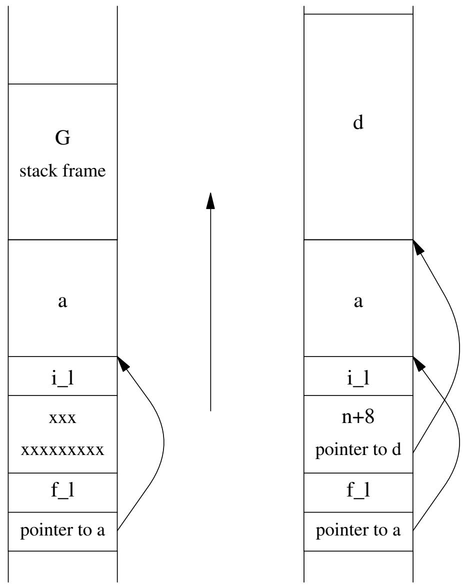


Figure 464.1: Storage for objects not having VLA type is allocated on block entry, plus storage for a descriptor for each object having VLA type. By the time `G` has been called, the declaration for `a` has been reached and storage allocated for it. After `G` returns, the declaration for `d` is reached and is storage allocated for it. The descriptor for `d` needs to include a count of the number of elements in one of the array dimensions. This value is needed for index calculations and is not known at translation time. No such index calculations need to be made for `a`.

---

standard signed integer types

There are five *standard signed integer types*, designated as **`signed char`**, **`short int`**, **`int`**, **`long int`**, and **`long long int`**. 480

Usage

It is possible to specify many of the integer types, in C, using more than one sequence of keywords. Usage information on integer types is given elsewhere (see Table 1378.1).

---

signed integer corresponding unsigned integer

For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword **`unsigned`**) that uses the same amount of storage (including sign information) and has the same alignment requirements. 486

Usage

Usage information on integer type specifiers is given elsewhere (see Table 1378.1, which does not include uses of integer types specified via typedef names).

Table 486.1: Occurrence of objects having different width integer types (as a percentage of all integer types) for embedded source and the SPECint95 benchmark (separated by a forward slash, e.g.,embedded/SPECint95). Adapted from Engblom.[61]

	8 bits	16 bits	32 bits
unsigned	70.8/1.3	14.0/0.4	2.1/44.9
signed	2.7/0.0	9.4/0.3	1.0/53.1

---

floating types three real

There are three *real floating types*, designated as **`float`**, **`double`**, and **`long double`**.32) 497

Table 497.1: Occurrence of floating types in various declaration contexts (as a percentage of all floating types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
float	35.2	15.1	8.3	0.7	21.0	80.3
double	8.5	7.9	0.5	0.7	2.2	19.7
long double	0.0	0.0	0.0	0.0	0.0	0.0
Total	43.6	22.9	8.8	1.5	23.2	

515 The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. character types

Table 515.1: Occurrence of character types in various declaration contexts (as a percentage of all character types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
char	16.4	3.6	1.2	0.1	6.6	28.0
signed char	0.2	0.3	0.0	0.1	0.3	1.0
unsigned char	18.1	10.6	0.4	0.8	41.2	71.1
Total	34.7	14.6	1.5	1.0	48.2	

517 An *enumeration* comprises a set of named integer constant values. enumeration set of named constants

Usage

A study by Gravley and Lakhotia[74] looked at ways of automatically deducing which identifiers, defined as object-like macros denoting an integer constant, could be members of the same enumerated type. The heuristics used to group identifiers were based either on visual clues (block of **#define**s bracketed by comments or blank lines), or the value of the macro body (consecutive values in increasing or decreasing numeric sequence; bit sequences were not considered). 1931 macro object-like

The 75 header files analyzed contained 1,225 macro definitions, of which 533 had integer constant bodies. The heuristics using visual clues managed to find around 55 groups (average size 8.9 members) having more than one member, the value based heuristic found 60 such groups (average size 6.7 members).

519 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. integer types

Table 519.1: Occurrence of integer types in various declaration contexts (as a percentage of those all integer types appearing in all of these contexts). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**typedef**	Member	Total
char	1.8	0.4	0.1	0.0	0.7	3.1
signed char	0.0	0.0	0.0	0.0	0.0	0.1
unsigned char	2.0	1.2	0.0	0.1	4.6	7.9
short	0.7	0.3	0.0	0.0	0.4	1.4
unsigned short	2.3	0.8	0.1	0.1	3.2	6.5
int	28.4	10.6	4.2	0.1	6.4	49.7
unsigned int	5.6	3.6	0.3	0.1	4.2	13.8
long	3.0	1.2	0.1	0.1	0.8	5.1
unsigned long	4.8	1.9	0.2	0.1	2.1	9.1
enum	0.9	0.9	0.4	0.4	0.8	3.3
Total	49.6	20.8	5.4	0.9	23.2	

520 The integer and real floating types are collectively called *real types*. real types

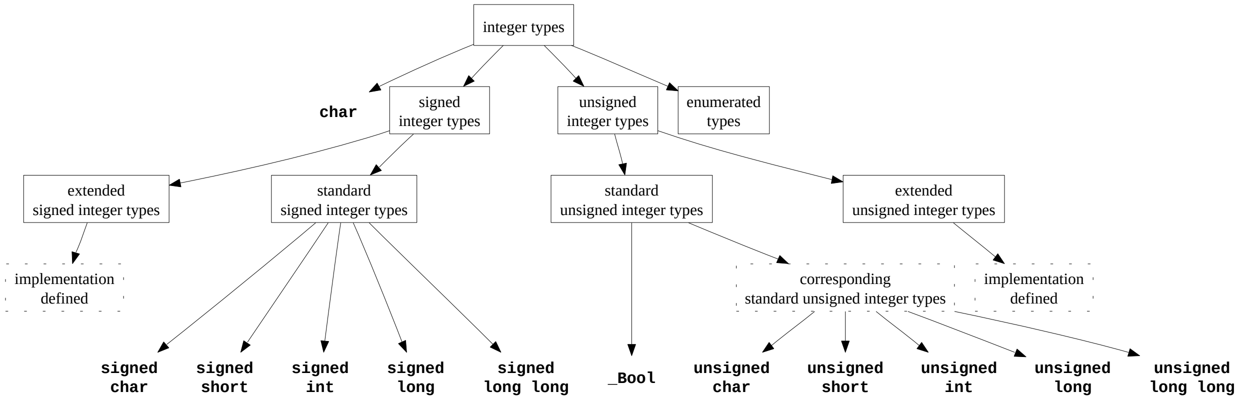


Figure 519.1: The integer types.

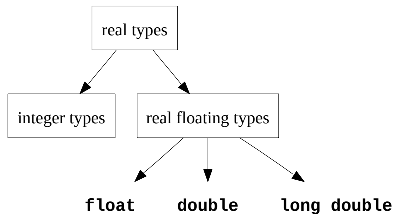


Figure 520.1: The real types.

---

arithmetic type

Integer and floating types are collectively called *arithmetic types*. 521

---

void is incomplete type

The **void** type comprises an empty set of values; 523

Usage

Information on keyword usage is given elsewhere (see Table 539.1, Table 758.1, Table 788.1, Table 1003.1, Table 1005.1, and Table 1134.1).

---

derived type

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows: 525

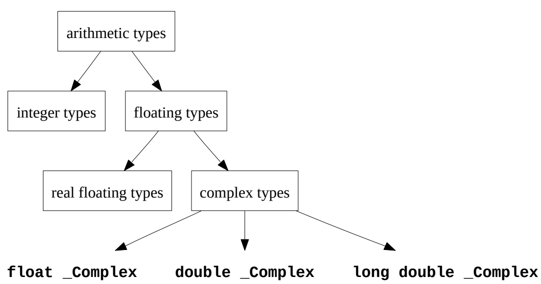


Figure 521.1: The arithmetic types.

Table 525.1: Occurrence of derived types in various declaration contexts (as a percentage of all derived types appearing in all of these contexts, e.g., `int **ap[2]` is counted as two pointer types and one array type). Based on the translated form of this book's benchmark programs.

Type	Block Scope	Parameter	File Scope	**`typedef`**	Member	Total
*	30.4	37.6	3.1	0.8	5.6	77.5
array	3.3	0.0	4.4	0.0	3.0	10.8
`struct`	3.7	0.1	2.4	2.3	2.6	11.2
`union`	0.2	0.0	0.0	0.1	0.2	0.5
Total	37.7	37.8	10.0	3.3	11.3	

527 Array types are characterized by their element type and by the number of elements in the array.

Table 527.1: Occurrence of arrays declared to have the given element type (as a percentage of all objects declared to have an array type). Based on the translated form of this book's benchmark programs.

Element Type	%	Element Type	%
`char`	17.2	**`struct`** *	3.7
`struct`	16.6	**`unsigned int`**	2.7
`float`	14.6	**`enum`**	2.5
other-types	10.4	**`unsigned short`**	2.0
`int`	8.5	**`float`** []	1.9
`const char`	8.0	**`const char`** * **`const`**	1.3
`char` *	5.1	**`short`**	1.1
`unsigned char`	4.4		

530 — A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type. structure type sequentially allocated objects

Usage

Usage information on the number of members in structure and union types and their types is given elsewhere.

297 limit members in struct/union
1403 struct member type

533 A function type is characterized by its return type and the number and types of its parameters.

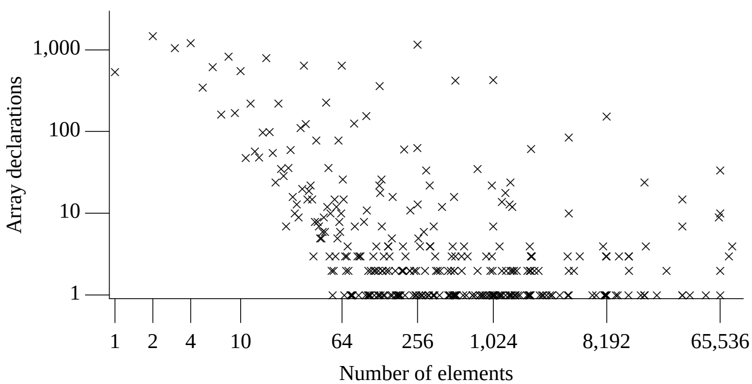


Figure 527.1: Number of arrays defined to have a given number of elements. Based on the translated form of this book's benchmark programs.

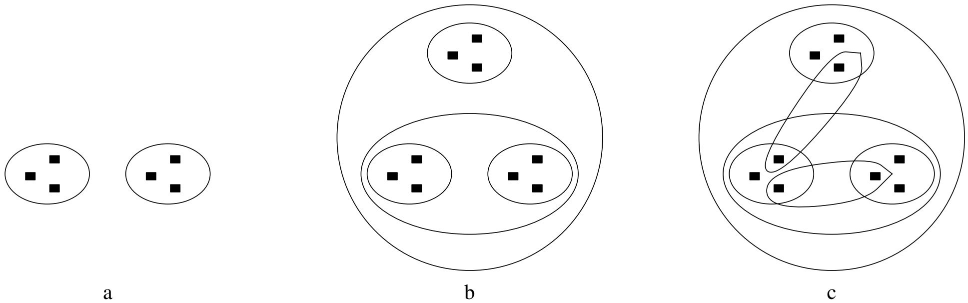


Figure 530.1: Three examples of possible member clusterings. In (a) there are two independent groupings, (b) shows a hierarchy of groupings, while in (c) it is not possible to define two C structure types that share a subset of common member (some other languages do support this functionality). The member c, for instance, might be implemented as a pointer to the value, or it may simply be duplicated in two structure types.

Usage

Usage information on function return types is given elsewhere (see Table 1005.1) as is information on parameters (see Table 1831.1).

---

pointer type
referenced type

— A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*. 539

Table 539.1: Occurrence of objects declared using a given pointer type (as a percentage of all objects declared to have a pointer type). Based on the translated form of this book's benchmark programs.

Pointed-to Type	%	Pointed-to Type	%
struct	66.5	**struct** *	1.8
char	8.0	**int**	1.8
union	6.0	**const char**	1.3
other-types	5.5	**char** *	1.2
void	3.3	str | str	
unsigned char	2.6	**_double | _double**	
unsigned int	2.2	**_double | _double**	

---

unqualified type

Any type so far mentioned is an *unqualified type*. 554

Table 554.1: Occurrence of qualified types as a percentage of all (i.e., qualified and unqualified) occurrences of that kind of type (e.g., * denotes any pointer type, **struct** any structure type, and *array of* an array of some type). Based on the translated form of this book's benchmark programs.

Type Combination	%	Type Combination	%
array of **const**	26.7	**const** *	0.4
const integer-type	4.8	**const union**	0.3
const real-type	2.7	**volatile struct**	0.1
* **const**	2.6	**volatile** integer-type	0.1
const struct	2.4	* **volatile**	0.1

---

qualified type
versions of

Each unqualified type has several *qualified versions* of its type,[38)] corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers. 555

Table 555.1: Occurrence of type qualifiers on the outermost type of declarations occurring in various contexts (as a percentage of all type qualifiers on the outermost type in these declarations). Based on the translated form of this book's benchmark programs.

Type Qualifier	Local	Parameter	File Scope	`typedef`	Member	Total
`const`	18.5	4.3	50.8	0.0	1.2	74.8
`volatile`	1.6	0.1	3.0	0.1	20.4	25.2
`volatile const`	0.0	0.0	0.0	0.0	0.0	0.0
Total	20.1	4.4	53.8	0.1	21.6	

559 Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements.

pointer to qualified/unqualified types

570 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

object contiguous sequence of bytes

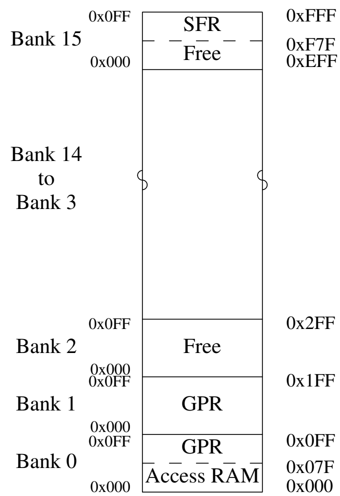


Figure 559.1: Data storage organization for the PIC18CXX2 devices[121] The 4,096 bytes of storage can be treated as a linear array or as 16 banks of 256 bytes (different instructions and performance penalties are involved). Some storage locations hold Special Function Registers (SFR) or General Purpose Registers (GPR). *Free* denotes storage that does not have a preassigned usage and is available for general program use.

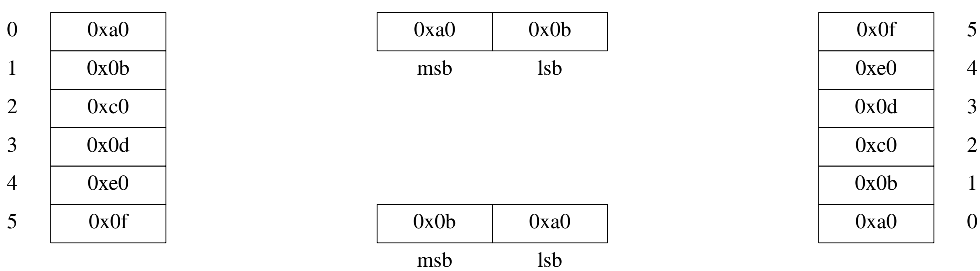


Figure 570.1: Developers who use little-endian often represent increasing storage locations going down the page. Developers who use big-endian often represent increasing storage locations going up the page. The value returned by an access to storage location *0*, using a pointer type that causes 16 bits to be read, will depend on the *endianness* of the processor.

Table 570.1: Byte order (indicated by the value of the digits) used by various processors for some integer and floating types, in different processor address spaces (all address spaces if none is specified).

Vendor	16-bit integer	32-bit integer	64-bit integer	32-bit float	64-bit float
AT&T 3B2		4321 (data space)/ 1234 (program space)			
DEC PDP–11	12	3412		3412 (F format)	78563412 (D format)
DEC VAX	12	1234	12345678	3412 (F format)	78563412 (D format)
NSC32016		1234 (data space)/ 4321 (program space)			

If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to 2^N-1 using a pure binary representation; 594

Table 594.1: Pattern of bits used to represent decimal numbers using various coding schemes.

Decimal	Binary	Gray code	111 biased	2–out-of–5
0	0000	0000	0111	00011
1	0001	0001	1000	00101
2	0010	0011	1001	00110
3	0011	0010	1010	01001
4	0100	0110	1011	01010
5	0101	0111	1100	01100
6	0110	0101	1101	10001
7	0111	0100	1110	10010
8	1000	1100	1111	10100
9	1001	1101		11000
10	1010	1111		
11	1011	1110		
12	1100	1010		
13	1101	1011		
14	1110	1001		
15	1111	1000		

sign bit representation

If the sign bit is one, the value shall be modified in one of the following ways: 610

operand convert automatically

Several operators convert operand values from one type to another automatically. 653

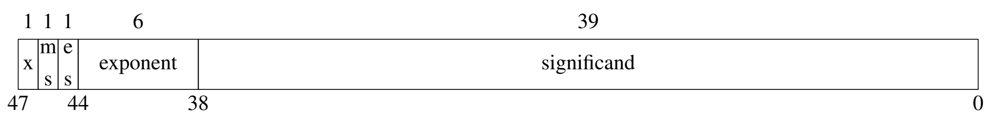


Figure 570.2: The Unisys A Series[171] uses the same representation for integer and floating-point types. For integer values bit 47 is unused, bit 46 represents the sign of the significand, bits 45 through 39 are zero, and bits 38 through 0 denote the value (a sign and magnitude representation). For floating values bit 47 represents the sign of the exponent and bits 46 through 39 represent the exponent (the representation for double-precision uses an additional word with bits 47 through 39 representing higher order-bits of the exponent and bits 38 through 0 representing the fractional portion of the significand).

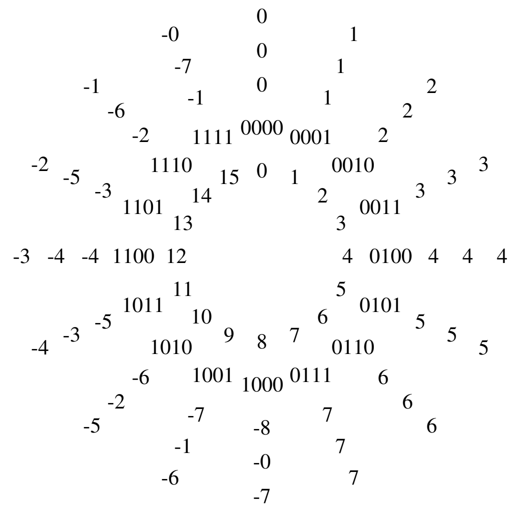


Figure 610.1: Decimal values obtained by interpreting a sequence of bits in various ways. From the inside out: unsigned, binary, two's complement, sign and magnitude, and one's complement.

Usage

Usage information on the cast operator is given elsewhere (see Table 1134.1).

Table 653.1: Occurrence of implicit conversions (as a percentage of all implicit conversions; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Converted to	Converted from	%	Converted to	Converted from	%
(**unsigned char**)	**_int**	33.0	(**int**)	**unsigned short**	1.9
(**unsigned short**)	**_int**	17.7	(**unsigned long**)	**_int**	1.8
(other-types)	other-types	11.3	(**unsigned int**)	**int**	1.7
(**short**)	**_int**	7.6	(**short**)	**int**	1.7
(**unsigned int**)	**_int**	5.1	(**enum**)	**_int**	1.3
(ptr-to)	ptr-to	4.7	(**unsigned long**)	**int**	1.2
(**char**)	**_int**	3.6	(**int**)	**char**	1.2
(ptr-to)	_ptr-to	2.9	(**int**)	**enum**	1.0
(**int**)	**unsigned char**	2.3			

675 These are called the *integer promotions*.[48] integer promotions

Table 675.1: Occurrence of integer promotions (as a percentage of all operands appearing in all expressions). Based on the translated form of this book's benchmark programs.

Original Type	%	Original Type	%
unsigned char	2.3	**char**	1.2
unsigned short	1.9	**short**	0.5

687 If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[50)]

710 Otherwise, the integer promotions are performed on both operands. arithmetic conversions integer promotions

Usage

Usage information on integer promotions is given elsewhere (see Table 675.1).

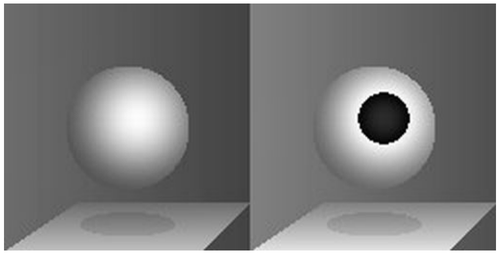


Figure 687.1: Illustration of the effect of integer addition wrapping rather than saturating. A value has been added to all of the pixels in the left image to increase the brightness, creating the image on the right. With permission from Jordán and Lotufo.[89]

arithmetic conversions integer types

Then the following rules are applied to the promoted operands: 711

Usage

Usage information on implicit conversions is given elsewhere (see Table 653.1).

lvalue converted to value

Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). 725

Usage

Usage information on the number of translation time references, in the source code, is given elsewhere (see Figure 1821.5, Figure 1821.6).

pointer permission to convert to integer

Any pointer type may be converted to an integer type. 754

Usage

Usage information on pointer conversions is given elsewhere (see Table 758.1 and Figure 1134.1).

pointer converted to pointer to different object or type

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. 758

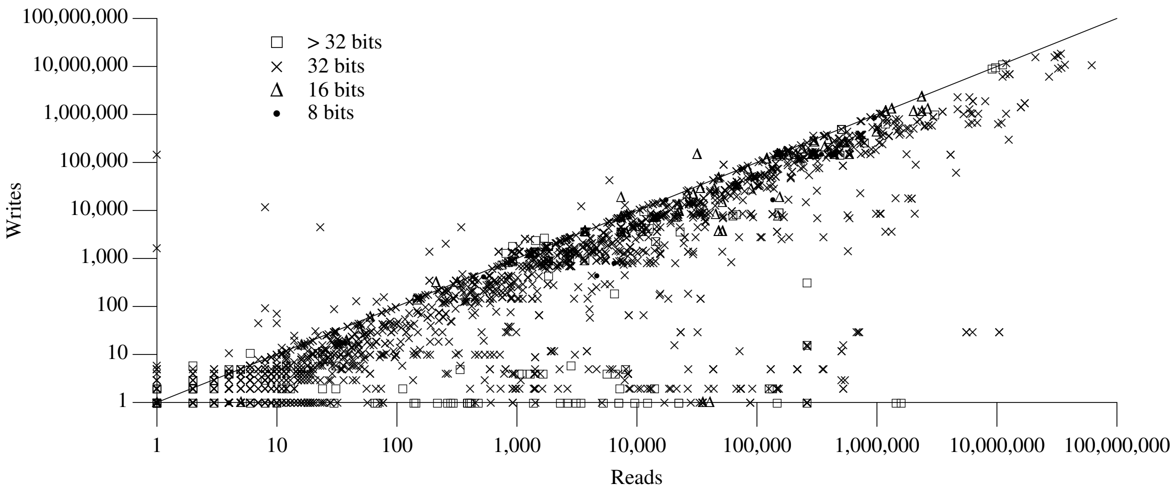


Figure 725.1: Execution-time counts of the number of reads and writes of the same object (declared in block or file scope, i.e., not allocated storage) for a subset of the MediaBench benchmarks; items above the diagonal indicate more writes than reads. Data kindly supplied by Caspi, based on his research.[31]

Table 758.1: Occurrence of implicit conversions that involve pointer types (as a percentage of all implicit conversions that involve pointer types). Based on the translated form of this book's benchmark programs.

To Type	From Type	%	To Type	From Type	%
(**struct** *)	**int**	44.0	(**void** *)	**int**	4.2
(function *)	**int**	18.4	(**unsigned char** *)	**int**	3.4
(**char** *)	**int**	7.9	(ptr-to *)	**int**	2.0
(**const char** *)	**int**	6.9	(**int** *)	**int**	1.9
(**union** *)	**int**	5.5	(**long** *)	**int**	1.1
(other-types *)	other-types *	4.7			

770

token syntax
preprocessing token syntax

```
token:
        keyword
        identifier
        constant
        string-literal
        punctuator
preprocessing-token:
        header-name
        identifier
        pp-number
        character-constant
        string-literal
        punctuator
        each non-white-space character that cannot be one of the above
```

Table 770.1: Mean percentage differences, compared to normal, in reading times (silent or aloud); the values in parenthesis are the range of differences. Adapted from Epelboim.[62]

Filler type	Surround	Fill-1	Fill-2	Unspaced
Shaded boxes (aloud)	4 (1–12)	—	3 (-2–9)	44 (25–60)
Digits (aloud)	26 (15–40)	26 (10–42)	—	42 (19–64)
Digits (silent)	40 (32–55)	41 (32–58)	—	52 (45–63)
Greek letters (aloud)	33 (20–47)	36 (23–45)	46 (33–57)	44 (32–53)
Latin letters (aloud)	55 (44–70)	—	74 (58–84)	43 (13–58)
Latin letters (silent)	66 (51–75)	75 (68–81)	—	45 (33–60)

Table 770.2: Number of expressions containing two binary operators (excluding any assignment operator, comma operator, function call operator, array access or member selection operators) having the specified spacing (i.e., no spacing, *no-space*, or one or more whitespace characters (excluding newline), *space*) between a binary operator and both of its operands. *High-Low* are expressions where the first operator of the pair has the higher precedence, *Same* are expressions where the both operators of the pair have the same precedence, *Low-High* are expressions where the first operator of the pair has the lower precedence. For instance, `x + y*z` is *space no-space* because there are one or more *space* characters either side of the addition operator and *no-space* either side of the multiplication operator, the precedence order is *Low-High*. Based on the visible form of the `.c` files.

	Total	High-Low	Same	Low-High
no-space	34,866	2,923	29,579	2,364
space no-space	4,132	90	393	3,649
space space	31,375	11,480	11,162	8,733
no-space space	2,659	2,136	405	118
total	73,032	16,629	41,539	14,864

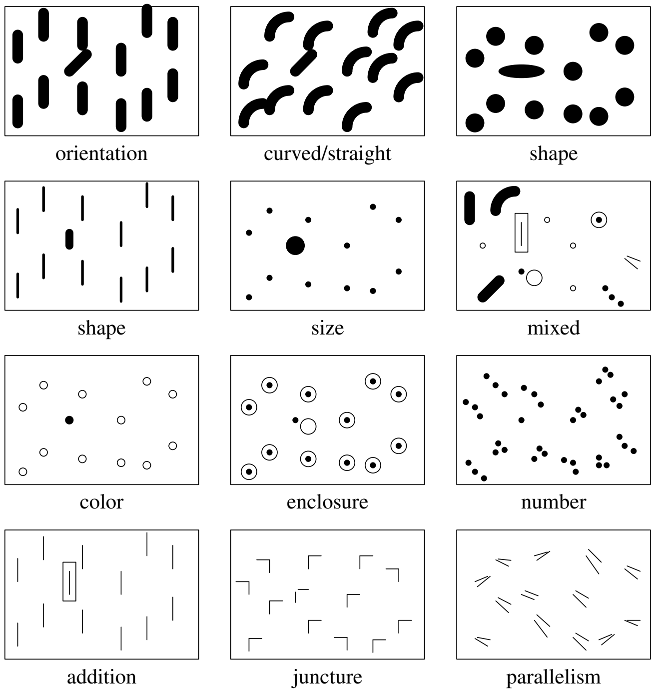


Figure 770.1: Examples of features that may be preattentively processed (parallel lines and the junction of two lines are the odd ones out). Adapted from Ware.[172]

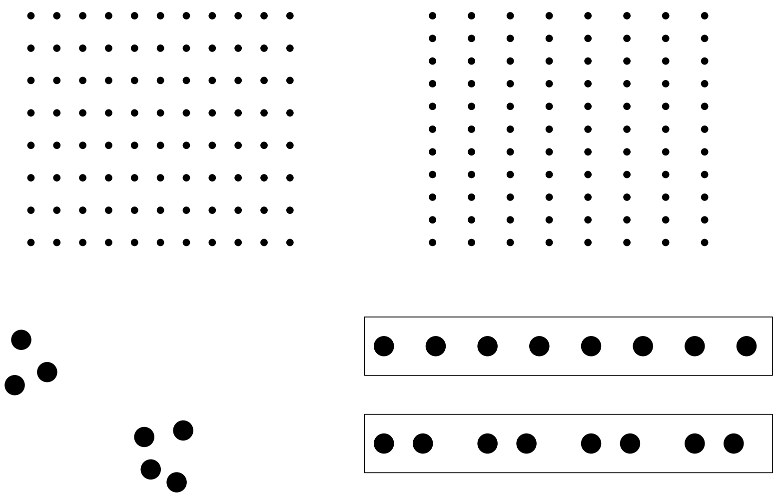


Figure 770.2: Proximity— the horizontal distance between the dots in the upper left image is less than the vertical distance, causing them to be perceptually grouped into lines (the relative distances are reversed in the upper right image).

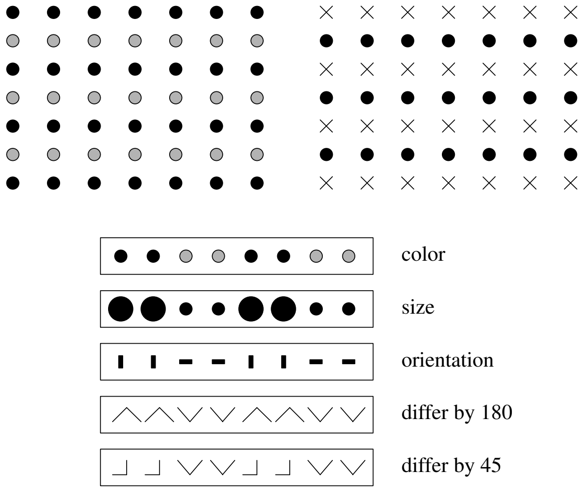


Figure 770.3: Similarity— a variety of dimensions along which visual items can differ sufficiently to cause them to be perceived as being distinct; rotating two line segments by 180° does not create as big a perceived difference as rotating them by 45°.

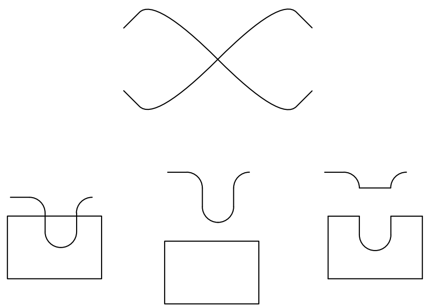


Figure 770.4: Continuity— upper image is perceived as two curved lines; the lower-left image is perceived as a curved line overlapping a rectangle rather than an angular line overlapping a rectangle having a piece missing (lower-right image).

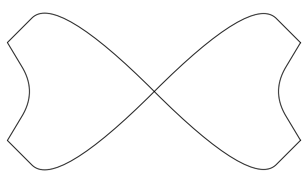


Figure 770.5: Closure— when the two perceived lines in the upper image of Figure 770.4 are joined at their end, the perception changes to one of two cone-shaped objects.

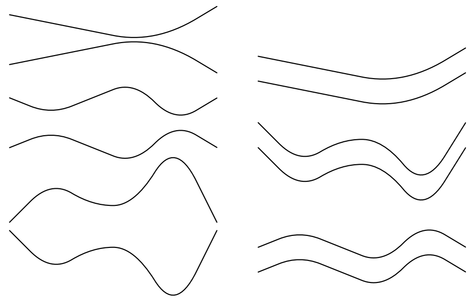


Figure 770.6: Symmetry and parallelism— where the direction taken by one line follows the same pattern of behavior as another line.

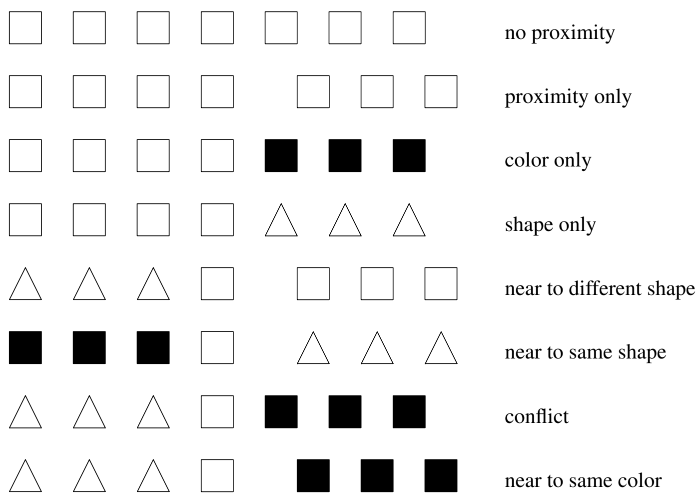


Figure 770.7: Conflict between proximity, color, and shape. Based on Quinlan.[140]

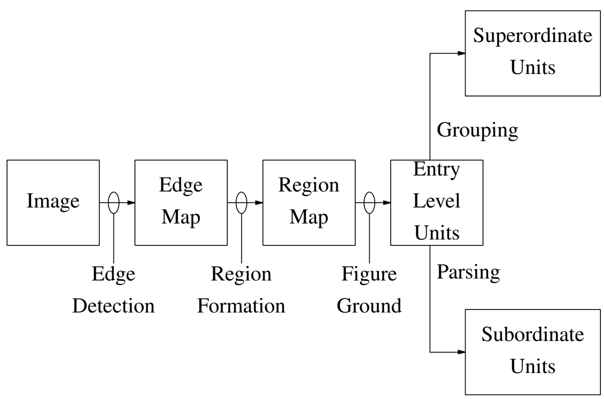


Figure 770.8: A flowchart of Palmer and Rock's[133] theory of perceptual organization.

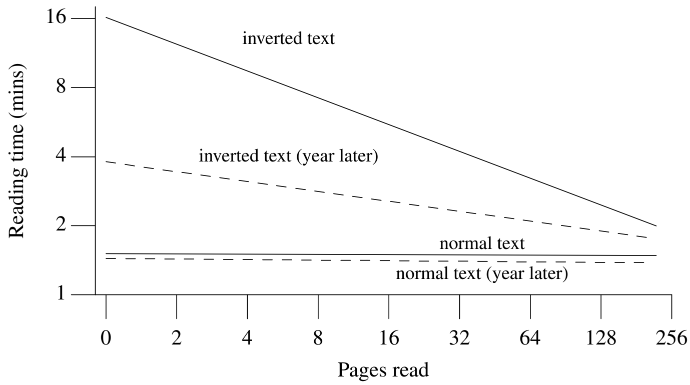


Figure 770.9: The time taken for subjects to read a page of text in a particular orientation, as they read more pages. Results are for the same six subjects in two tests more than a year apart. Based on Kolers.[100]

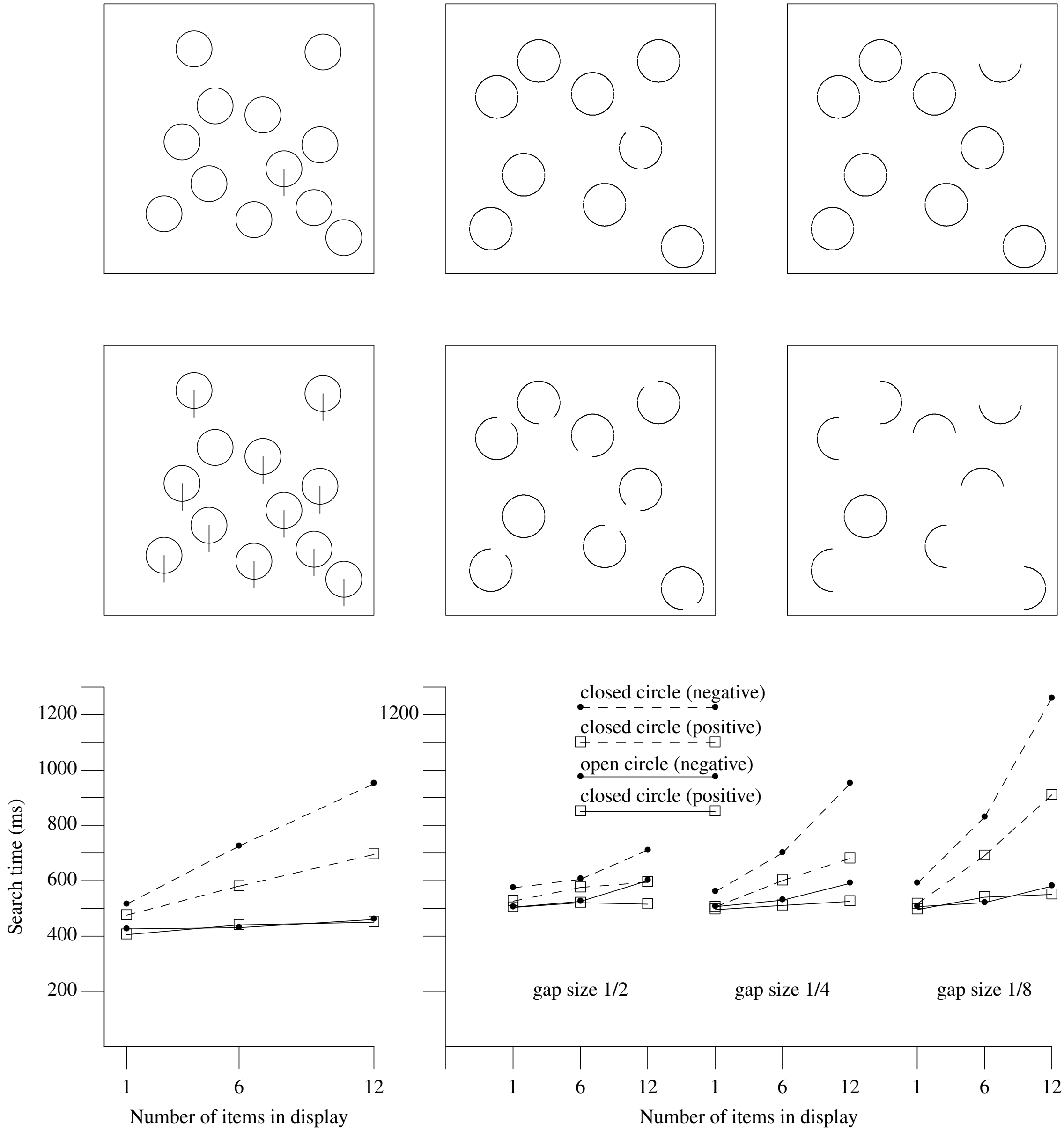


Figure 770.10: Examples of unique items among visually similar items. Those at the top include an item that has a distinguishing feature (a vertical line or a gap); those underneath them include an item that is missing this distinguishing feature. Graphs represent time taken to locate unique items (positive if it is present, negative when it is not present) when placed among different numbers of visibly similar distractors. Based on displays used in the study by Treisman and Sother.[166]

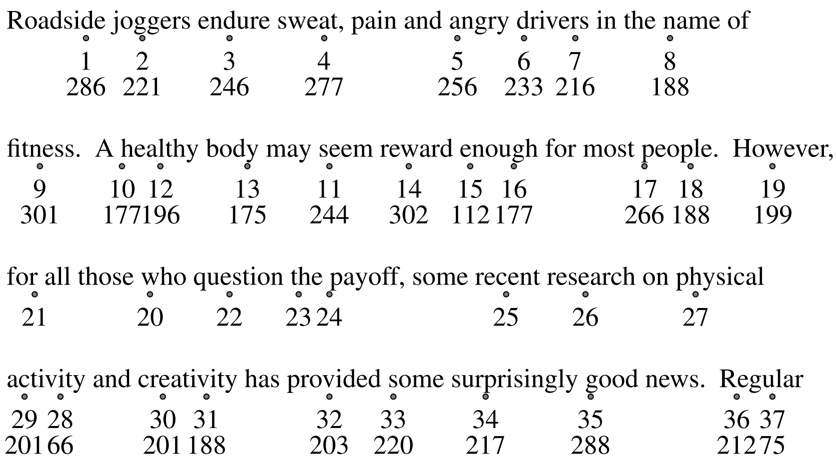


Figure 770.11: A passage of text with eye fixation position (dot under word), fixation sequence number, and fixation duration (in milliseconds) included. Adapted from Reichle, Pollatsek, Fisher, and Rayner[145] (timings on the third line are missing in the original).

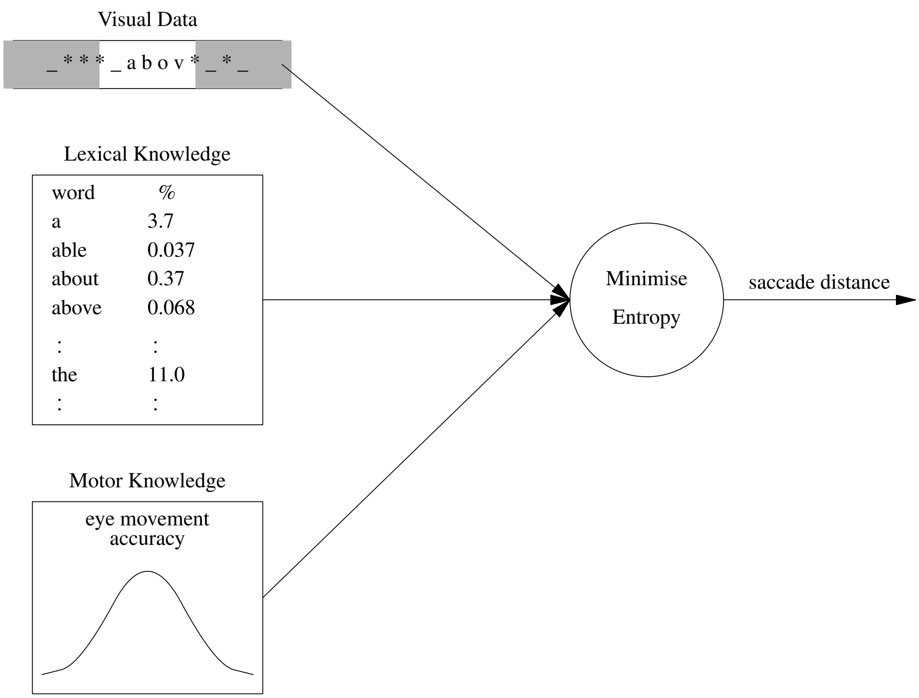


Figure 770.12: Mr. Chips schematic. The shaded region in the visual data is the parafoveal; in this region individual letters (indicated by stars) can only be distinguished from spaces (indicated by underscores). Based on Legge et al.[109]

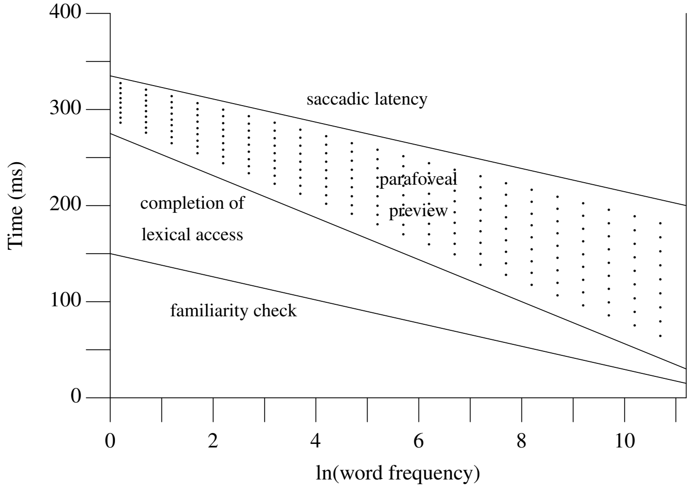


Figure 770.13: How preview benefit is affected by word frequency. The bottom line denotes the time needed to complete the familiarity check, the middle line the completion of lexical access, and the top line when the execution of the eye movement triggered by the familiarity check occurs. Based on Reichle, Pollatsek, Fisher, and Rayner.[145]

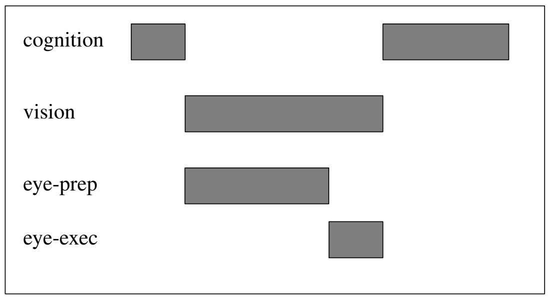


Figure 770.14: Example case of EMMA's control flow. Adapted from Salvucci.[147]

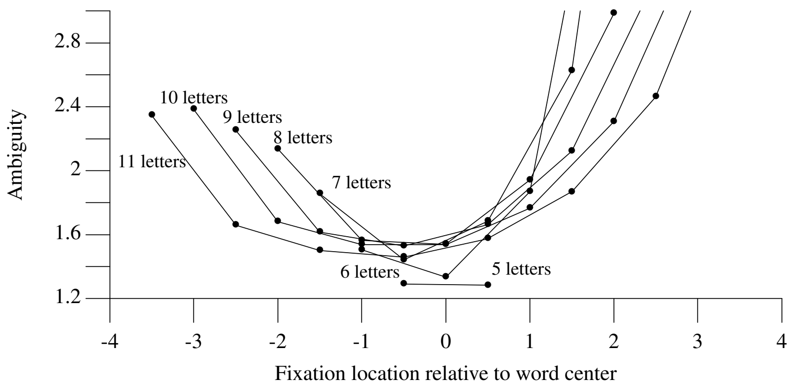


Figure 770.15: The ambiguity of patterns defined by the first and last letter and an interior letter pair, as a function of the position of the first letter of the pair. Plots are for different word lengths using the 65,000 words from CLAWS[107] (as used by the `aspell` tool). The fixation position is taken to be midway between the interior letter pair.

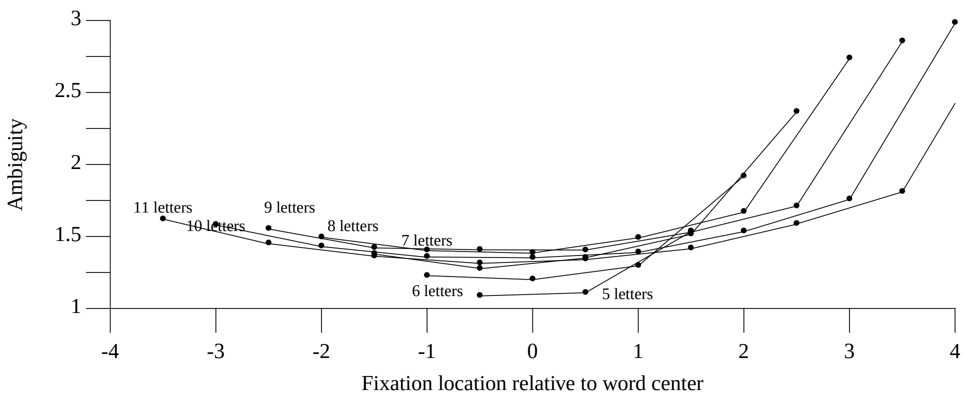


Figure 770.16: The ambiguity of source code identifiers, which can include digits as well as alphabetic characters. Plots are for different identifier lengths. A total of 344,000 identifiers from the visible form of the `.c` files were used.

translation phase 124 3

Usage

Table 770.3 shows the relative frequency of the different kinds of tokens in a source file (actual token count information is given elsewhere). Adding the percentages for *Preceded by Space* and *First on Line* (or followed by space and last on line) does not yield 100% because of other characters occurring in those positions. Some tokens occur frequently, but contribute a small percentage of the characters in the visible source (e.g., punctuators). Identifier tokens contribute more than 40% of the characters in the `.c` files, but only represent 28.5% of the tokens in those files.

Table 770.3: Occurrence of kinds of tokens in the visible form of the `.c` and `.h` files as a percentage of all tokens (value in parenthesis is the percentage of all non-white-space characters contained in those tokens), percentage occurrence (for `.c` files only) of token kind where it was preceded/followed by a space character, or starts/finishes a visible line. While comments are not tokens they are the only other construct that can contain non-white-space characters. While the start of a preprocessing directive contains two tokens, these are generally treated by developers as a single entity.

Token	% of Tokens in `.c` files	% of Tokens in `.h` files	% Preceded by Space	% Followed by Space	% First Token on Line	% Last Token on Line
punctuator	53.5 (11.4)	48.1 (7.5)	27.5	29.7	3.7	25.3
identifier	29.8 (43.4)	20.8 (30.6)	54.9	27.6	1.4	1.2
constant	6.9 (3.8)	21.6 (15.3)	70.3	4.4	0.1	1.6
keyword	6.9 (5.8)	5.4 (4.2)	79.9	82.5	10.3	3.6
comment	1.9 (31.0)	3.4 (40.3)	53.4	2.2	41.2	97.4
string-literal	1.0 (4.6)	0.8 (2.2)	59.9	5.7	0.7	8.0
pp-directive	0.9 (1.1)	4.9 (4.4)	4.7	78.4	0.0	18.2
header-name	0.0 (0.0)	0.0 (0.0)	–	–	–	–

---

preprocessing tokens white space separation

Preprocessing tokens can be separated by *white space*; 777

Usage

Table 770.3 shows the relative frequency of white space occurring before and after various kinds of tokens.

788

```
keyword: one of
    auto        enum        restrict    unsigned
    break       extern      return      void
    case        float       short       volatile
    char        for         signed      while
    const       goto        sizeof      _Bool
    continue    if          static      _Complex
```

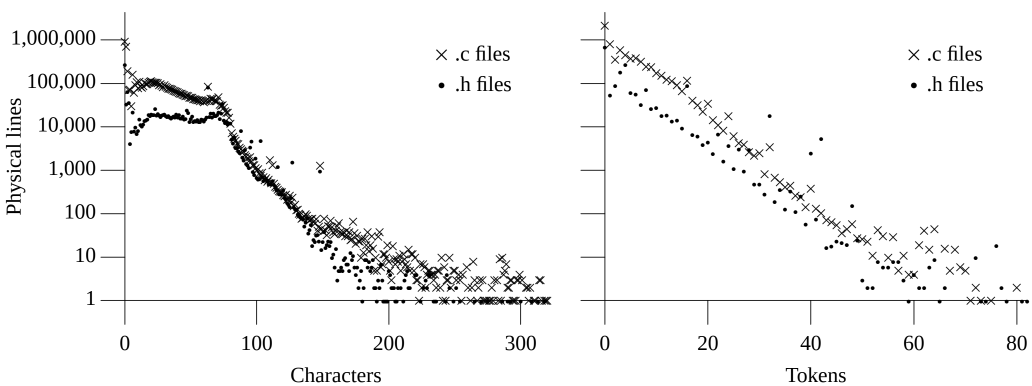


Figure 770.17: Number of physical lines containing a given number of non-white-space characters and tokens. Based on the visible form of the `.c` and `.h` files.

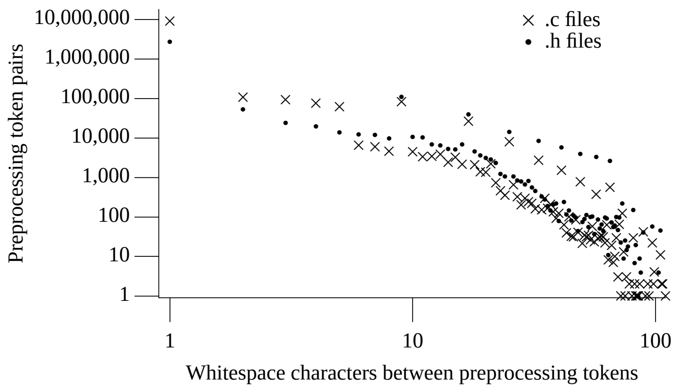


Figure 777.1: Number of *pp-token* pairs having the given number of white-space characters them (does do not include white space at the start of a line— i.e., indentation white space, and end-of-line is not counted as a white-space character). Based on the visible form of the `.c` and `.h` files.

default	**inline**	**struct**	**_Imaginary**
do	**int**	**switch**	
double	**long**	**typedef**	
else	**register**	**union**	

Usage

Usage information on preprocessor directives is given elsewhere (see Table 1854.1).

Table 788.1: Occurrence of keywords (as a percentage of all keywords in the respective suffixed file) and occurrence of those keywords as the first and last token on a line (as a percentage of occurrences of the respective keyword; for `.c` files only). Based on the visible form of the `.c` and `.h` files.

Keyword	.c Files	.h Files	% Start of Line	% End of Line	Keyword	.c Files	.h Files	% Start of Line	% End of Line
if	21.46	15.63	93.60	0.00	**const**	0.94	0.80	35.50	0.30
int	11.31	13.40	47.00	5.30	**switch**	0.75	0.77	99.40	0.00
return	10.18	12.23	94.50	0.10	**extern**	0.61	0.71	99.60	0.40
struct	8.10	10.33	38.90	0.30	**register**	0.59	0.64	95.00	0.00
void	6.24	10.27	28.70	18.20	**default**	0.54	0.58	99.90	0.00
static	6.04	8.07	99.80	0.60	**continue**	0.49	0.33	91.30	0.00
char	4.90	5.08	30.50	0.20	**short**	0.38	0.28	16.00	1.00
case	4.67	4.81	97.80	0.00	**enum**	0.20	0.27	73.70	1.80
else	4.62	3.30	70.20	42.20	**do**	0.20	0.25	87.30	21.30
unsigned	4.17	2.58	46.80	0.10	**volatile**	0.18	0.17	50.00	0.00
break	3.77	2.44	91.80	0.00	**float**	0.16	0.17	54.00	0.70
sizeof	2.23	2.24	11.30	0.00	**typedef**	0.15	0.09	99.80	0.00
long	2.23	1.49	10.10	1.70	**double**	0.14	0.08	53.60	3.10
for	2.22	1.06	99.70	0.00	**union**	0.04	0.06	63.30	6.20
while	1.23	0.95	85.20	0.10	**signed**	0.02	0.01	27.20	0.00
goto	1.23	0.89	94.10	0.00	**auto**	0.00	0.00	0.00	0.00

identifier syntax

identifier:

identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:

nondigit
universal-character-name
other implementation-defined characters

nondigit: one of

_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

```
#        <      . >

#                    13
#                     0
#                     1

                 (              [],
                      *            )
{
    ,
           ;

*           =          ;
      =      (        );
   (         >             )
   {
   *          =            ;
   }

   {
       ( =0;   <          ;  ++)
      {
         ((        [ ] < '0') ||
          (        [ ] > '9'))
         {
         *            =            ;
         }
      }
   }
}
```

```
 include  string h

 define MAX_CNUM_LEN
 define VALID_CNUM
 define INVALID_CNUM

int chk_cnum_valid char cust_num
                   int  cnum_status

int i
    cnum_len

 cnum_status VALID_CNUM
cnum_len strlen cust_num
if  cnum_len   MAX_CNUM_LEN

    cnum_status INVALID_CNUM

else

   for  i    i   cnum_len  i

      if   cust_num i
           cust_num i

          cnum_status INVALID_CNUM
```

```
#include <string.h>

#define v1 13
#define v2  0
#define v3  1

int v4(char v5[],
       int *v6)
{
int v7,
    v8;

*v6=v2;
v8=strlen(v5);
if (v8 > v1)
   {
   *v6=v3;
   }
else
   {
   for (v7=0; v7 < v8; v7++)
      {
      if ((v5[v7] < '0') ||
          (v5[v7] > '9'))
         {
         *v6=v3;
         }
      }
   }
}
```

Figure 792.1: The same program visually presented in three different ways; illustrating how a reader's existing knowledge of words can provide a significant benefit in comprehending source code. By comparison, all the other tokens combined provide relatively little information. Based on an example from Laitinen.[103]

Table 792.1: Mean comprehension rating and mean number of ideas recalled from passage (standard deviation is given in parentheses). Adapted from Bransford and Johnson.[22]

	No Topic Given	Topic Given After	Topic Given Before	Maximum Score
Comprehension	2.29 (0.22)	2.12 (0.26)	4.50 (0.49)	7
Recall	2.82 (0.60)	2.65 (0.53)	5.83 (0.49)	18

Table 792.2: Break down of issues considered applicable to selecting an identifier spelling.

	Visual	Acoustic	Semantic	Miscellaneous
Memory	Idetic memory	Working memory is sound based	Proper names, LTM is semantic based	spelling, cognitive studies, Learning
Confusability	Letter and word shape	Sounds like	Categories, metaphor	Sequence comparison
Usability	Careful reading, visual search	Working memory limits, pronounceability	interpersonal communication, abbreviations	Cognitive resources, typing

Table 792.3: Estimates of the number of speakers each language (figures include both native and nonnative speakers of the language; adapted from Ethnologue volume I, SIL International). Note: Hindi and Urdu are essentially the same language, Hindustani. As the official language of Pakistan, it is written right-to-left in a modified Arabic script and called Urdu (106 million speakers). As the official language of India, it is written left-to-right in the Devanagari script and called Hindi (469 million speakers).

Rank	Language	Speakers (millions)	Writing direction	Preferred word order
1	Mandarin Chinese	1,075	left-to-right also top-down	SVO
2	Hindi/Urdu	575	see note	see note
3	English	514	left-to-right	SVO
4	Spanish	425	left-to-right	SVO
5	Russian	275	left-to-right	SVO
6	Arabic	256	right-to-left	VSO
7	Bengali	215	left-to-right	SOV
8	Portuguese	194	left-to-right	SVO
9	Malay/Indonesian	176	left-to-right	SVO
10	French	129	left-to-right	SVO
11	German	128	left-to-right	SOV
12	Japanese	126	left-to-right	SOV

Table 792.4: Percentage of identifiers in one program having the same spelling as identifiers occurring in various other programs. First row is the total number of identifiers in the program and the value used to divide the number of shared identifiers in that column). Based on the visible form of the `.c` files.

	gcc	idsoftware	linux	netscape	openafs	openMotif	postgresql
	46,549	27,467	275,566	52,326	35,868	35,465	18,131
gcc	—	2	9	6	5	3	3
idsoftware	5	—	8	6	5	4	3
linux	1	0	—	1	1	0	0
netscape	5	3	8	—	5	7	3
openafs	6	4	12	8	—	3	5
openMotif	4	3	6	11	3	—	3
postgresql	9	5	12	11	10	6	—

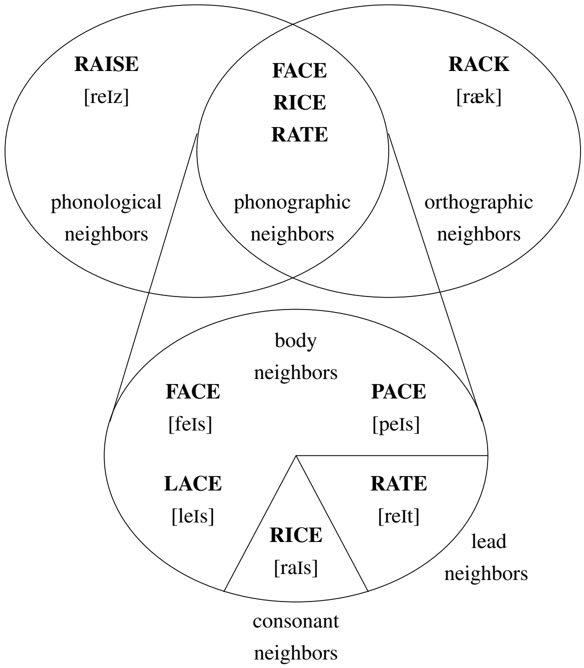


Figure 792.2: Example of the different kinds of lexical neighborhoods for the English word *RACE*. Adapted from Peereman and Content.[137]

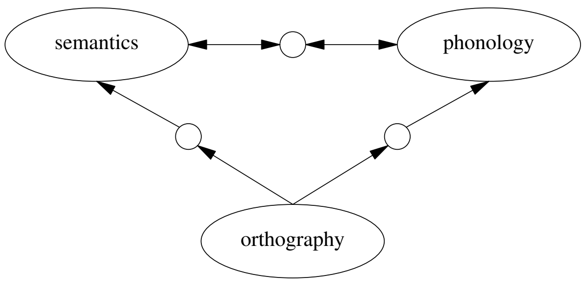


Figure 792.3: Triangle model of word recognition. There are two routes to both semantics and phonology, from orthography. Adapted from Harm.[78]

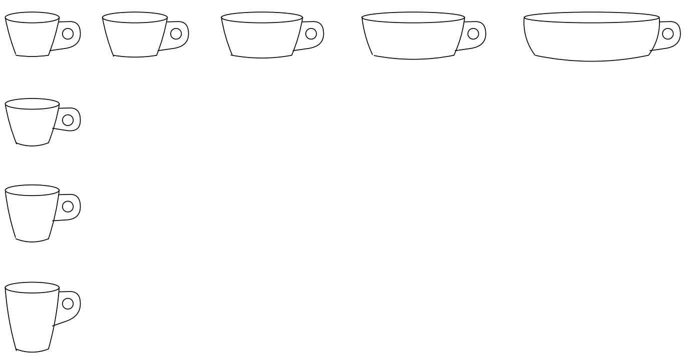


Figure 792.4: Cup- and bowl-like objects of various widths (ratios 1.2, 1.5, 1.9, and 2.5) and heights (ratios 1.2, 1.5, 1.9, and 2.4). Adapted from Labov.[102]

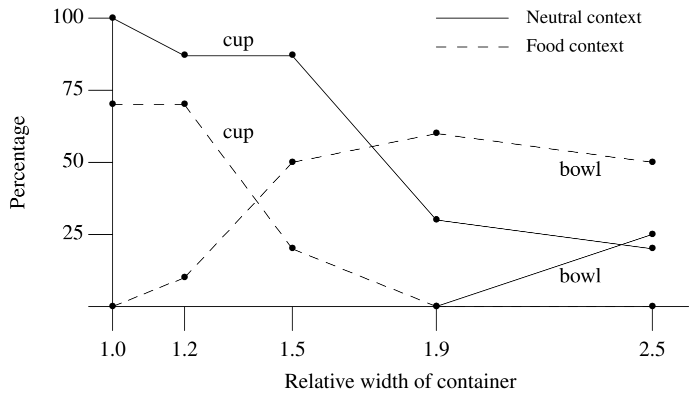


Figure 792.5: The percentage of subjects who selected the term *cup* or *bowl* to describe the object they were shown (the paper did not explain why the figures do not sum to 100%). Adapted from Labov.[102]

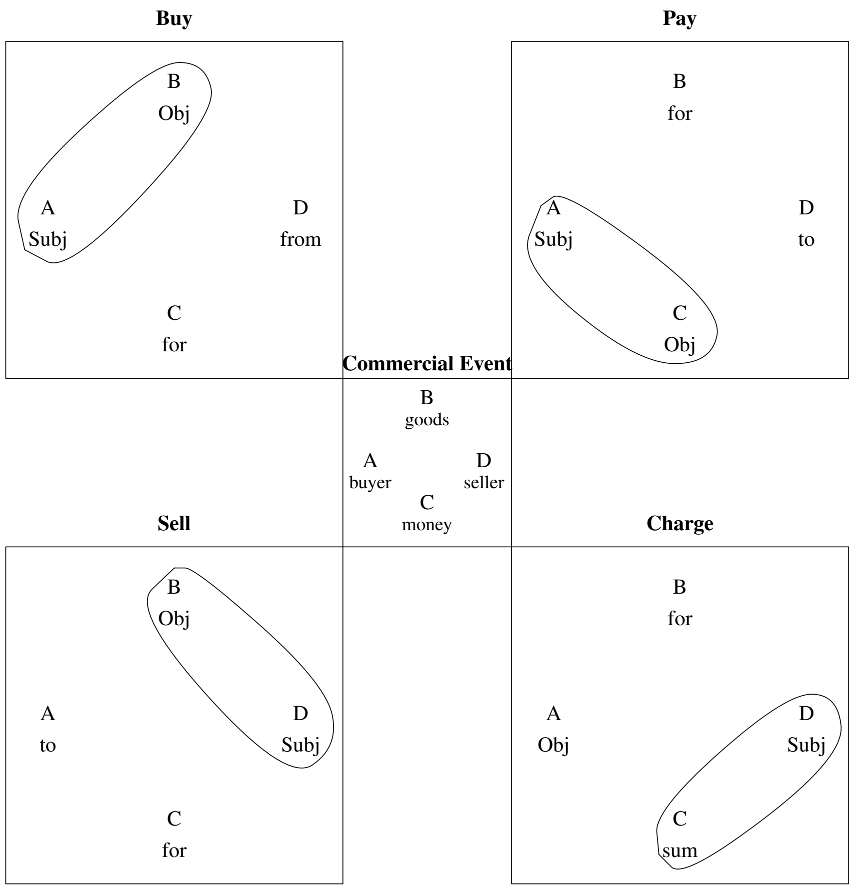


Figure 792.6: A commercial event involving a buyer, seller, money, and goods; as seen from the buy, sell, pay, or charge perspective. Based on Fillmore.[67]

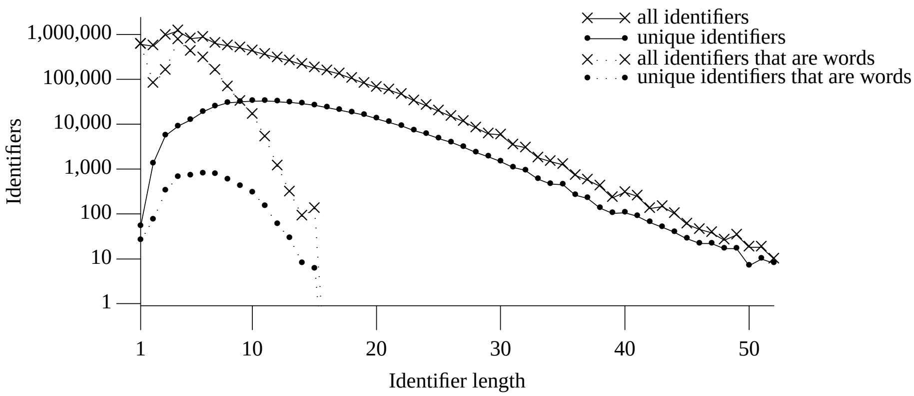


Figure 792.7: Number of identifiers (unique and all) of different length in the visible form of the `.c` files. Any identifier whose spelling appeared in the `aspell` 65,000 word dictionary was considered to be a word.

Table 792.5: Occurrence of identifier declarations in various scopes and name spaces (as a percentage of all identifiers within the scope/name space in the visible form of the `.c` files; unique identifiers are in parentheses) containing particular character sequences (the phrase *spelled using upper-case letters* is usually taken to mean that no lower-case letters are used, i.e., digits and underscore are included in the possible set of characters; for simplicity and accuracy the set of characters omitted are listed).

	no lower-case	no upper-case	no underscore	no digits	only first character upper-case
file scope objects	0.8 (1.0)	80.3 (79.1)	29.6 (25.4)	87.3 (85.7)	5.2 (5.7)
block scope objects	1.3 (1.8)	91.9 (81.3)	79.9 (58.9)	96.3 (93.0)	1.3 (3.1)
function parameters	0.1 (0.4)	94.2 (82.9)	88.6 (67.4)	96.8 (94.8)	1.4 (2.9)
function definitions	0.2 (0.2)	59.0 (62.1)	27.1 (24.1)	87.1 (86.4)	29.9 (27.3)
struct/union members	0.5 (0.8)	78.5 (71.8)	65.7 (51.3)	93.2 (91.4)	12.0 (14.2)
function declarations	0.7 (0.5)	55.5 (57.1)	27.3 (26.5)	88.7 (87.5)	32.4 (30.1)
tag names	5.7 (6.6)	60.7 (63.8)	25.6 (21.6)	88.1 (85.9)	18.4 (14.5)
`typedef` names	14.0 (17.0)	37.0 (33.5)	45.0 (40.4)	89.7 (89.3)	39.8 (37.4)
enumeration constants	55.8 (56.0)	10.8 (10.6)	16.0 (15.0)	79.9 (77.9)	32.1 (32.0)
label names	27.2 (48.1)	69.2 (47.4)	70.8 (65.6)	67.4 (46.3)	2.2 (2.3)
macro definitions	78.4 (79.9)	4.9 (5.0)	15.5 (13.0)	70.9 (69.3)	13.1 (11.1)
macro parameters	19.8 (20.4)	77.6 (68.7)	96.0 (83.6)	94.2 (90.7)	1.4 (5.0)

Table 792.6: Number of people using particular types of writing system for the top 50 world languages in terms of number of speakers. Literacy rates from UNESCO based on typical countries for each language (e.g., China, Egypt, India, Spain). Adapted from Cook.[47]

Total languages out of 50	Speakers (millions)	Readers (millions, based on illiteracy rates)
Character-based systems— 8 (all Chinese) + Japanese	1,088	930
Syllabic systems— 13 (mostly in India) + Japanese, Korean	561	329
Consonantal systems— 4 (two Arabic) + Urdu, Persian	148	no figures available
Alphabetic systems— 21 (worldwide)	1,572	1,232

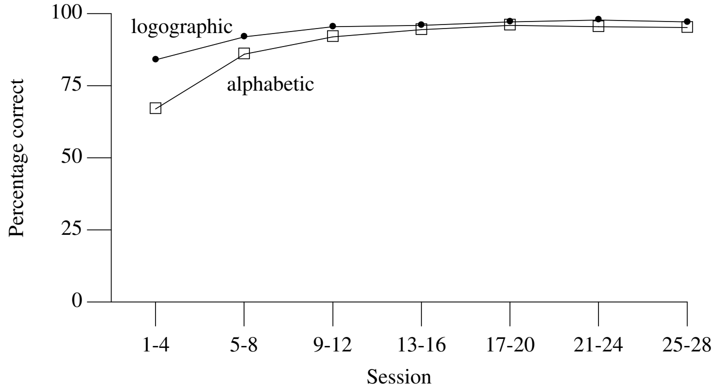


Figure 792.8: Improvement in word-recognition performance with number of sessions (most sessions consisted of 16 blocks of 16 trials). Adapted from Muter and Johns.[127]

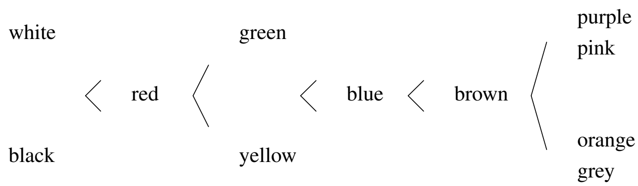


Figure 792.9: The original Berlin and Kay[16] language color hierarchy. The presence of any color term in a language, implies the existence, in that language, of all terms to its left. Papuan Dani has two terms (black and white), while Russian has eleven. (Russian may also be an exception in that it has two terms for blue.)

Table 792.7: Known number of languages commonly using a particular word order. Based on Comrie.[44]

Common order	Languages	Example
None	no figures	Sanskrit
SOV	180	Turkish “Hansan ököz-ü aldι” ⇒ “Hassan ox bought”
SVO	168	English “The farmer killed the duckling”
VSO	37	Welsh “Lladdodd y ddraig y dyn” ⇒ “killed the dragon the man”
VOS	12	Malagasy “Nahita ny mpianatra ny vehivavy” ⇒ “saw the student the woman”
OVS	5	Hixkaryana “Toto yahosi-ye kamara” ⇒ “man it-grabbed-him jaguar”
OSV	2	Apurinã none available

Table 792.8: The 12 *tenses* of English (actually three tenses and four aspects). Adapted from Celce-Murcia.[34]

	Simple	Perfect	Progressive	Perfect progressive
Present	write/writes walk/walks	has/have written has/have walked	am/is/are writing am-is/are walking	has/have been writing has/have been walking
Past	wrote walked	had written had walked	was/were writing was/were walking	had been writing had been walking
Future	will write will walk	will have written will have walked	will be writing will be walking	will have been writing will have been walking

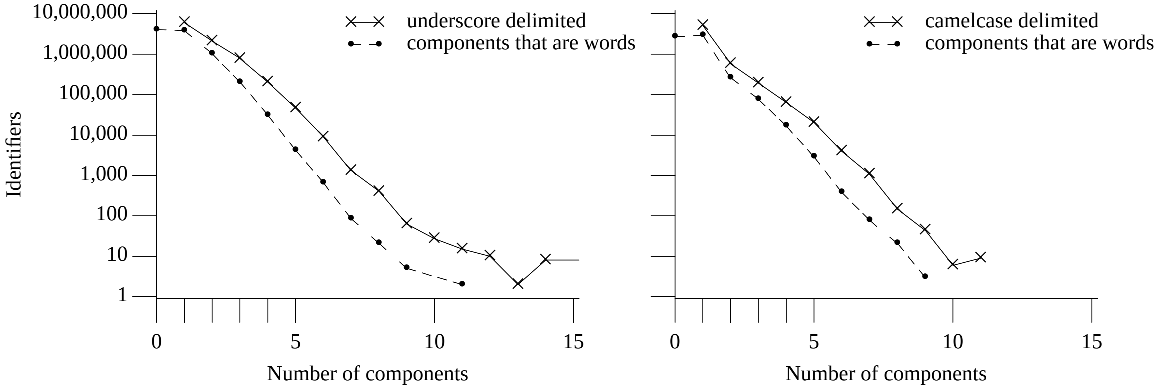


Figure 792.10: Number of identifiers containing a given number of *components*. In the left graph a component is defined as a character sequence delimited by one or more underscore characters, `_`, the start of the identifier, or its ending, e.g., the identifier `big_blk_proboscis` is considered to contain three components, one of which is a word. In the right graph a component is any sequence of lower-case letters, a sequence of two or more upper-case characters (i.e., the sequence is terminated by one or more digit characters or a letter having opposite case), or an upper-case character followed by a sequence of lower-case letters (this form of identifier might be said to be written in *camelCase*). For instance, the identifier `bigBlk4proboscis` is considered to contain three components, one of which is a word. A word is defined by the contents of the `ispell` 65,000 word list (this means, for instance, that the character sequence proboscis is not considered to be a word). Based on the visible form of the `.c` files.

Table 792.9: Probability of an adjective occurring at a particular position relative to other adjectives. Adapted from Celce-Murcia.[34]

determiner	option	size	shape	condition	age	color	origin	noun
	0.80	0.97	0.66	0.79	0.85	0.77	1.0	
an	ugly	big	round	chipped	old	blue	French	vase

Table 792.10: Subcategories of determiners. Adapted from Celce-Murcia.[34]

Predeterminers	Core determiners	Post determiners
qualifiers: *all, both, half*, etc. fractions: *such a, what a*, etc. multipliers: *double, twice, three times*, etc.	articles: *a, an, the*, etc. possessives: *my, our*, etc. demonstratives: *this, that*, etc.	cardinal numbers: *one, two*, etc. ordinal numbers: *first, second*, etc. general ordinals: *next, last, another*, etc.
	quantifiers: *some, any, no, each, every, either, neither, enough*, etc.	quantifiers: *many, much, (a) few (a) little, several, more, less most, least*, etc. phrasal quantifiers: *a great deal, of, a lot of, a good number of*, etc.

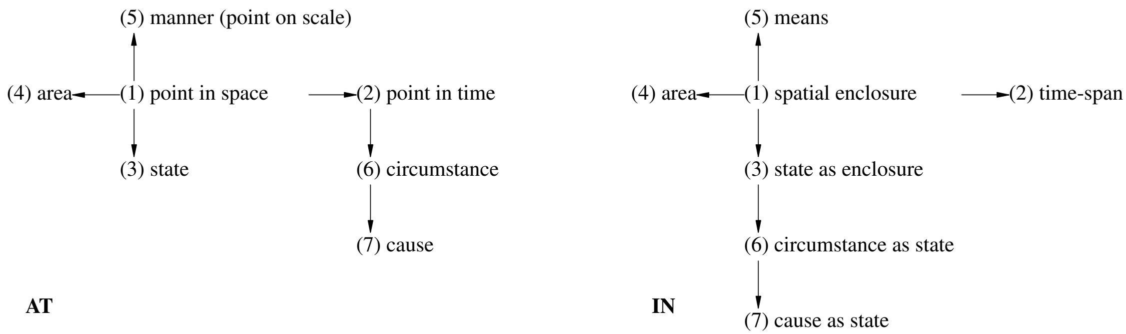


Figure 792.11: Examples, using "**at**" and "**in**" of extensions of prepositions from physical to mental space. Adapted from Dirven.[56]

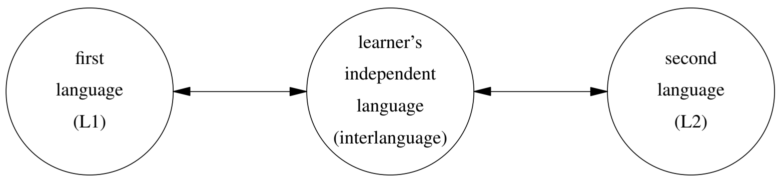


Figure 792.12: A learner's independent language— *interlanguage*. This language changes as learners go through the various stages of learning a new language. It represents the rules and structures invented by learners, which are influenced by what they already know, as they acquire knowledge and proficiency in a new language.

Table 792.11: Example words and total number of all mistakes for particular spelling patterns (*–C–* denotes any consonant). Adapted from Sloboda.[150]

Spelling pattern	similar phonologically	mistakes made	dissimilar phonologically	mistakes made
-ent -ant	clement clemant	46	convert convart	1
-ce -se	promice promise	9	polich polish	1
w- wh-	weight wheight	3	sapely shapely	1
-er -or	paster pastor	7	parret parrot	6
-le -el	hostle hostel	11	assits assist	1
-ayed -aid	sprayed spraid	18	slayer slair	0
-ea- -ee-	deamed deemed	24	dearth deerth	3
-CC- -C-	deppress depress	33	preessed pressed	0
-ancy -ency	currancy currency	27	corractly correctly	0
-al -el	rival rivel	13	livas lives	2

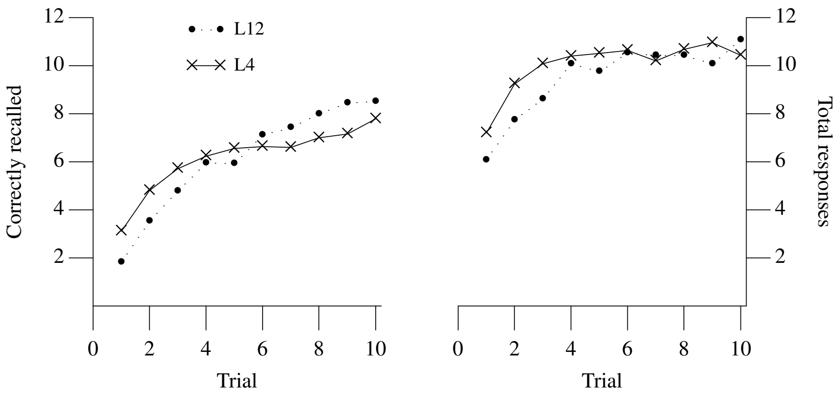


Figure 792.13: Mean correct recall scores and mean number of responses (correct and incorrect) for 10 trials. Adapted from Horowitz.[84]

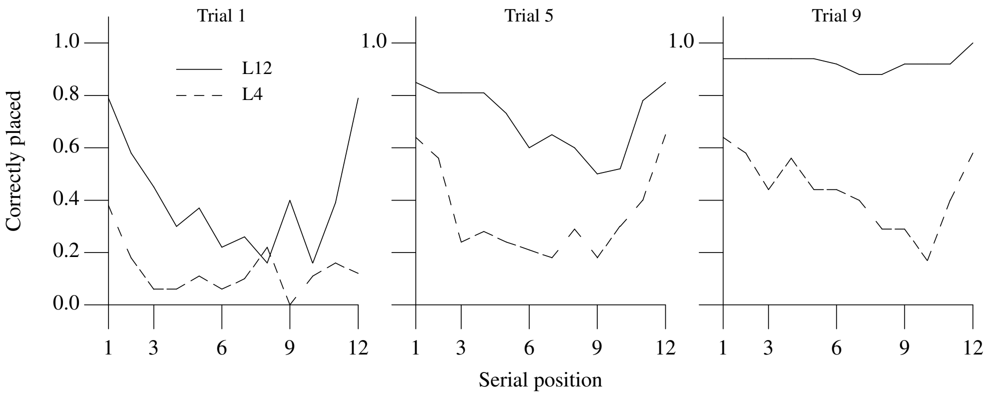


Figure 792.14: Percentage of correct orderings as a function of the trigram position within the list learned for three different trials. Adapted from Horowitz.[84]

Table 792.12: Mean number of each kind of information recalled in each condition (maximum score: 48). Adapted from Cohen.[42]

	Name	Occupation	Possession
Nonword	18.6	37.1	16.5
Word	23.6	37.0	30.4

Table 792.13: Breakdown of 52,963 spelling mistakes in 25 million typed words. Adapted from Pollock and Zamora.[139]

Kind of Mistake	Percentage Mistakes
omission	34
insertion	27
substitution	19
transposition	12.5
more than one	7.5

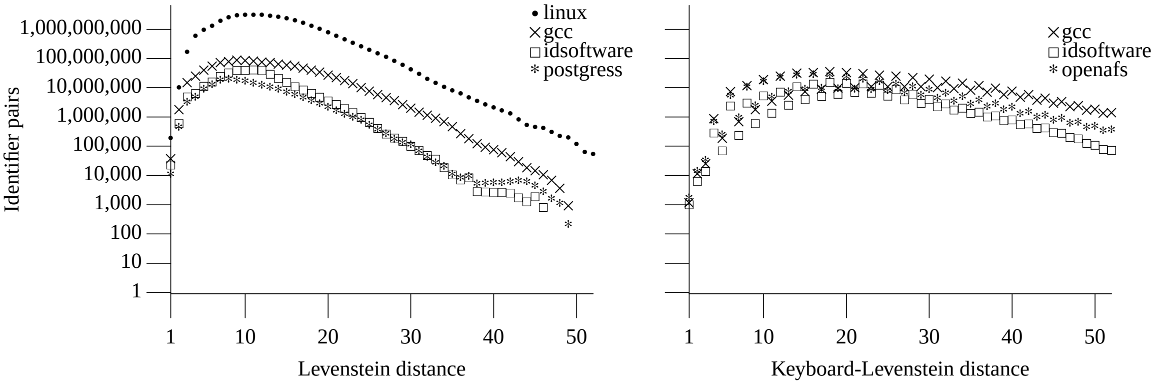


Figure 792.15: Number of identifiers having a given Levenstein distance from all other identifiers occurring in the visible form of the `.c` files of individual programs (i.e., identifiers in gcc were only compared against other identifiers in gcc). The *keyboard-levenstein* distance was calculated using a weight of 1 when comparing characters on immediately adjacent keyboard keys and a weight of 2 for all other cases (the result was normalized to allow comparison against unweighted Levenstein distance values).

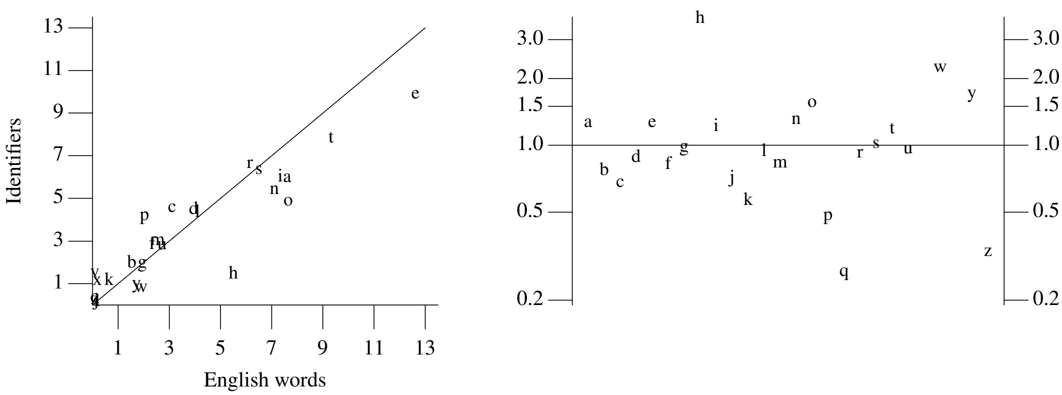


Figure 792.16: Occurrence of alphabetic letters in English text[152] and identifier names (based on the visible form of the `.c` files; all letters mapped to lowercase). Left graph: the letter percentage occurrence as (x, y) coordinates; right graph: the ratio of dividing the English by the identifier letter frequency (i.e., letters above the line are more common in English text than in identifiers; two letters outside the range plotted are $v = 0.0588$ and $x = 0.165$).

Table 792.14: Mean number of spelling mistakes for high/low frequency words with regular/irregular spellings. Adapted from Brown.[23]

	High Frequency Regular Spelling	Low Frequency Regular Spelling	High Frequency Irregular Spelling	Low Frequency Irregular Spelling
Native speaker	0.106	4.213	0.596	7.319
Second language	0.766	7.383	2.426	9.255
Example	cat, paper	fen, yak	of, one	tsetse, ghoul

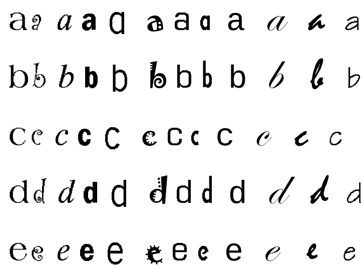


Figure 792.17: A number of different glyphs (different fonts are used) for various characters.

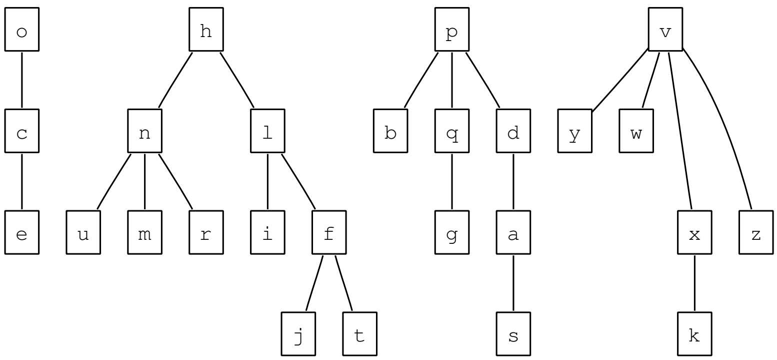


Figure 792.18: Similarity hierarchy for English letters. Adapted from *Lost reference*.[?]

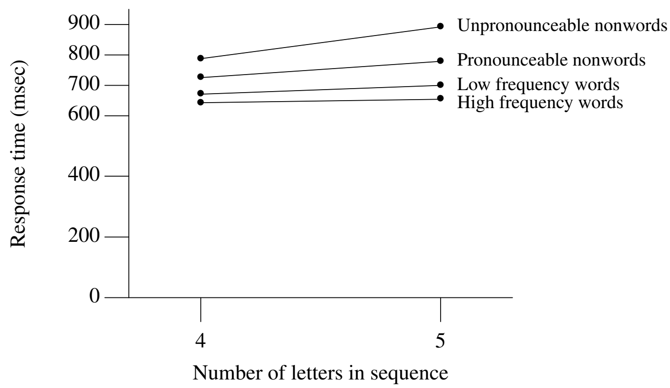


Figure 792.19: Response time to match two letter sequences as being identical. Adapted from Chambers and Foster.[35]

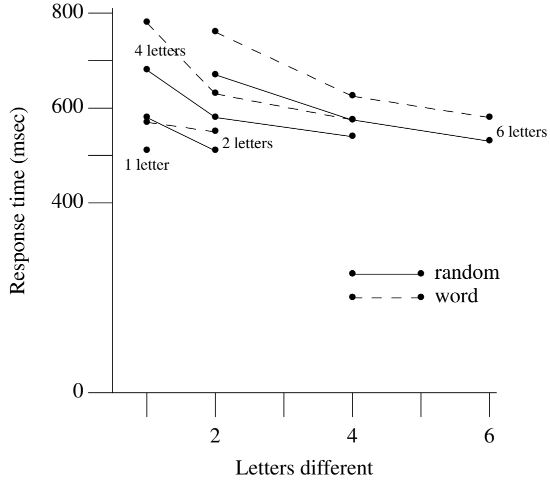


Figure 792.20: Time taken (in milliseconds) to match a pair of letter sequences as being identical— for different number of letters in the sequence and number of positions in the sequence containing a nonmatching letter. Adapted from Eichelman.[58]

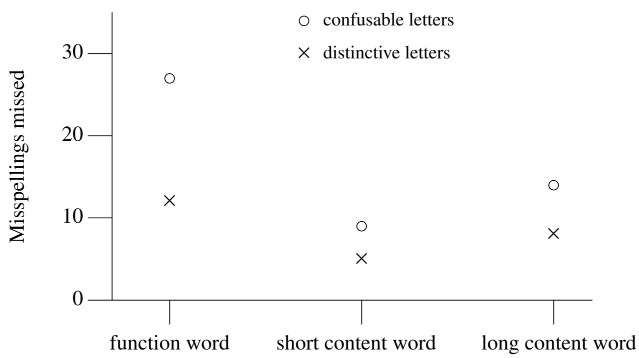


Figure 792.21: Percentage of misspellings not detected for various kinds of word. Adapted from Paap, Newsome, and Noel.[132]

Table 792.15: Response time (in milliseconds) to fail to match two letter sequences. Right column is average response time to match identical letter sequences. Columns are ordered by which letter differed between letter sequences. Adapted from Chambers and Foster.[35]

	All Letters	First Letter	Third Letter	Fifth Letter	Same Response
Words	677	748	815	851	747
Pronounceable nonwords	673	727	844	886	873
Unpronounceable nonwords	686	791	1,007	1,041	1,007

Table 792.16: Proportion of spelling errors detected (after arcsin transform was applied to the results). Adapted from Monk and Hulme.[125]

	Same Lowercase Word Shape	Different Lowercase Word Shape	Same Mixedcase Word Shape	Different Mixedcase Word Shape
Letter deleted	0.554	0.615	0.529	0.517
Letter substituted	0.759	0.818	0.678	0.680

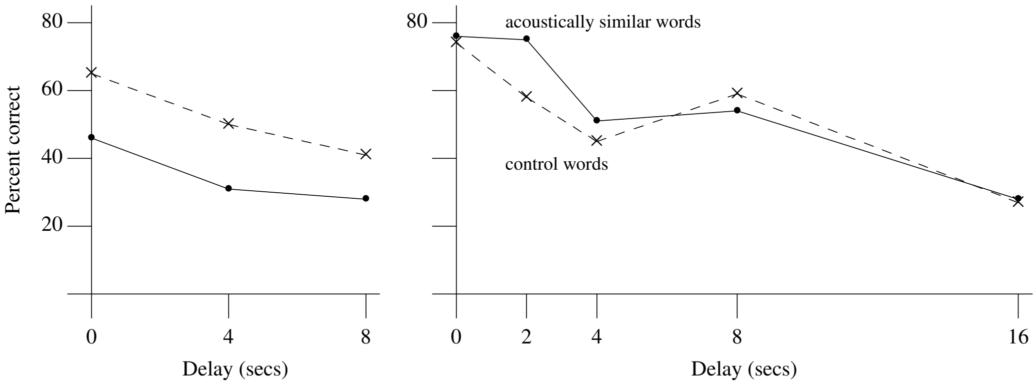


Figure 792.22: Rate of forgetting of visually presented lists of four words containing the same (solid line) or different vowels (dashed line); left graph. Rate for two lists, one containing three acoustically similar words (solid line) and the other five control words (dashed line); right graph. Adapted from Baddeley.[9]

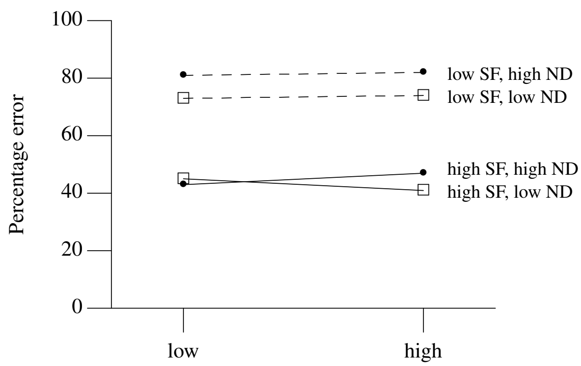


Figure 792.23: Error rate at low and high neighborhood frequency. Stimulus (drug name) frequency (SF), neighborhood density (ND). Adapted from Lambert, Chang, and Gupta.[105]

Table 792.17: Classification of recall errors for acoustically similar (AS), acoustically dissimilar (AD) pairs of letters. *Semi-transpose* refers to the case where, for instance, *PB* is presented and *BV* is recalled (where *V* does not appear in the list). *Other* refers to the case where pairs are both replaced by completely different letters. Adapted from Conrad.[46]

Number Intervening Letters	Transpose (AS)	Semi-transpose (AS)	Other (AS)	Transpose (AD)	Semi-transpose (AD)	Other (AD)	Total
0	797	446	130	157	252	207	1,989
1	140	112	34	13	33	76	408
2	31	23	16	2	18	56	146
3	12	20	12	1	5	23	73
4	0	4	1	0	2	7	14
Total	890	605	193	173	310	369	2,630

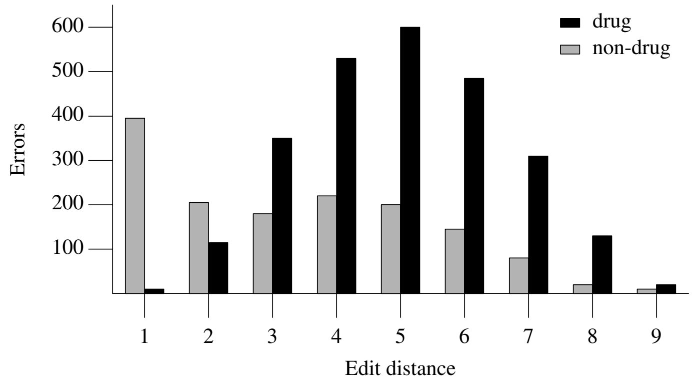


Figure 792.24: Number of substitution errors having a given edit distance from the correct response. Grey bars denote non-drug-name responses, while black bars denote responses that are known drug names. Based on Lambert, Chang, and Gupta.[105]

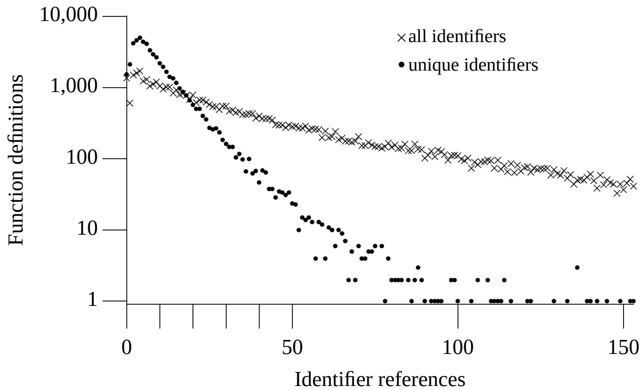


Figure 792.25: Number of identifiers referenced within individual function definitions. Based on the translated form of this book's benchmark programs.

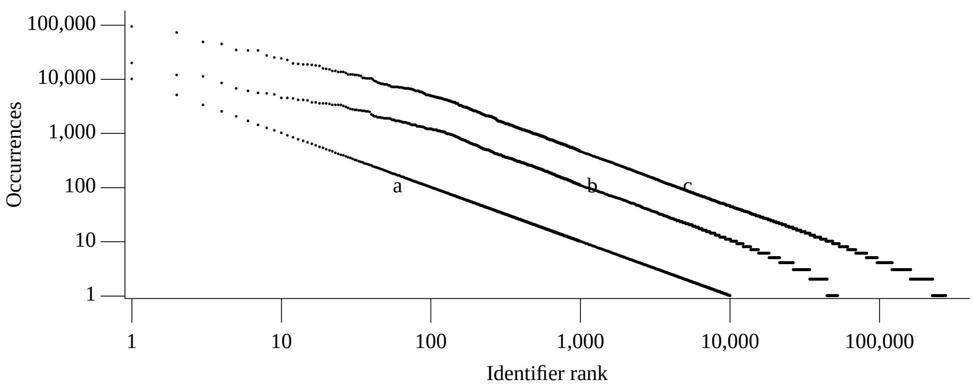


Figure 792.26: Identifier rank (based on frequency of occurrence of identifiers having a particular spelling) plotted against the number of occurrences of the identifier in the visible source of (b) Mozilla, and (c) Linux 2.4 kernel; (a) is a distribution following Zipf's law with the most common item occurring 10,000 times. Every identifier is represented by a dot. Also see Figure 1896.4.

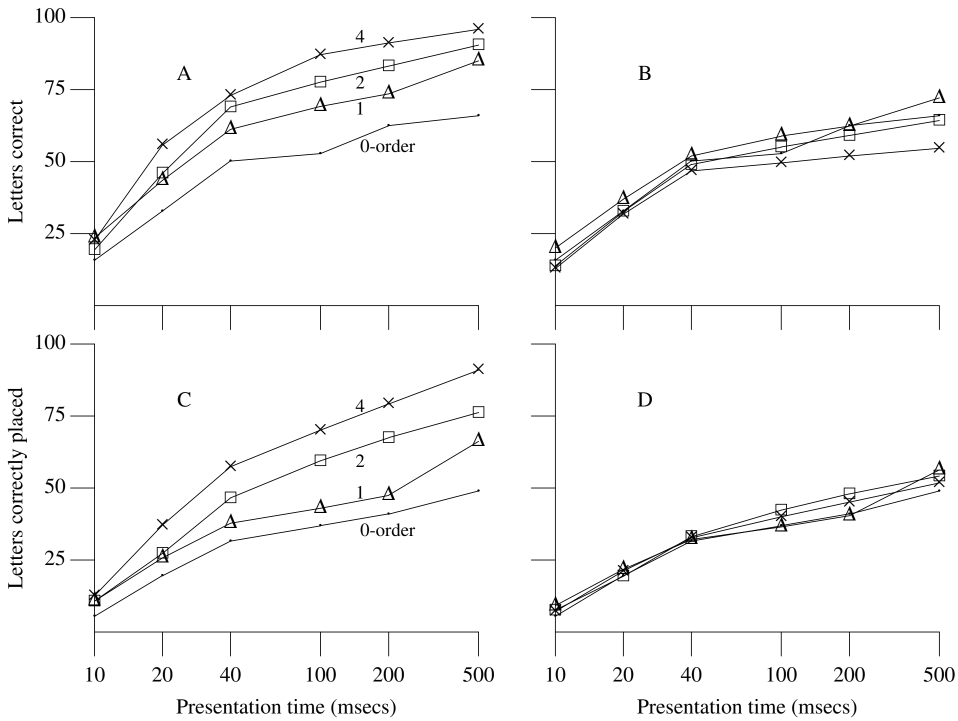


Figure 792.27: Number of correct letters regardless of position (A), and number of correct letters placed in the correct position (C). Normalizing for information content, the corresponding results are (B) and (D), respectively. Plotted lines denote 0-, 1-, 2-, and 4-order approximations to English words (see Table 792.18). Adapted from Miller, Bruner, and Postman.[122]

Table 792.18: Examples of nonwords. The 0-order words were created by randomly selecting a sequence of equally probable letters, the 1-order words by weighting the random selection according to the probability of letters found in English words, the 2-order words by weighting the random selection according to the probability of a particular letter following the previous letter in the nonword (for English words), and so on. Adapted from Miller[122]).

0-order	1-order	2-order	4-order
YRULPZOC	STANUGOP	WALLYLOF	RICANING
OZHGPMTJ	VTYEHULO	RGERARES	VERNALIT
DLEGQMNW	EINOAASE	CHEVADNE	MOSSIANT
GFUJXZAQ	IYDEWAKN	NERMBLIM	POKERSON
WXPAUJVB	RPITCQET	ONESTEVA	ONETICUL
VQWVBIFX	OMNTOHCH	ACOSUNST	ATEDITOL
CVGJCDHM	DNEHHSNO	SERRRTHE	APHYSTER
MFRSIWZE	RSEMPOIN	ROCEDERT	TERVALLE

Table 792.19: Words that make up 19 of the 46 words beginning with the English */gl/* of the monomorphemic vocabulary (Note: The others are: globe, glower, glean, glib, glimmer, glimpse, gloss, glyph, glib, glide, glitter, gloss, glide, glissade, glob, globe, glut, glean, glimmer, glue, gluten, glutton, glance, gland, glove, glad, glee, gloat, glory, glow, gloom, glower, glum, glade, and glen). Adapted from Magnus.[114]

Concept Denoted	Example Words
Reflected or indirect light	glare, gleam, glim, glimmer, glint, glisten, glister, glitter, gloaming, glow
Indirect use of the eyes	glance, glaze(d), glimpse, glint
Reflecting surfaces	glacé, glacier, glair, glare, glass, glaze, gloss

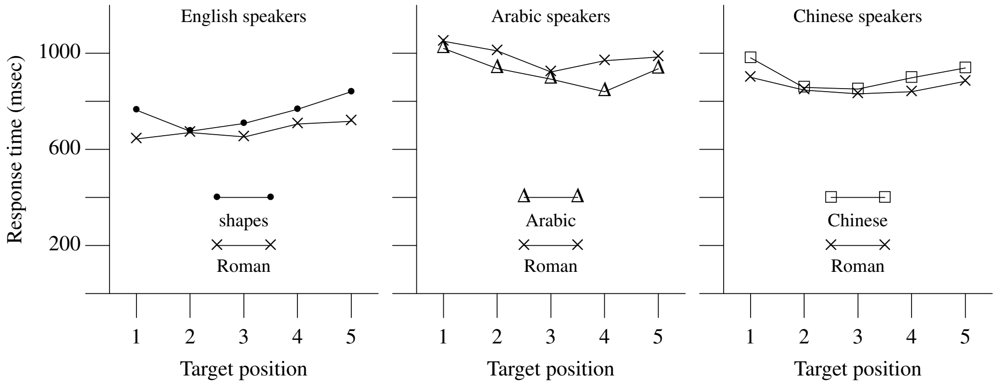


Figure 792.28: Mean response time (in milliseconds) for correct target detection as a function of the position of the match within the character sequence. Adapted from Green and Meara.[75]

```
#include <string.h>

#define MAXIMUM_CUSTOMER_NUMBER_LENGTH 13
#define VALID_CUSTOMER_NUMBER           0
#define INVALID_CUSTOMER_NUMBER         1

int check_customer_number_is_valid(char possibly_valid_customer_number[],
                                   int *customer_number_status)
{
int customer_number_index,
    customer_number_length;

*customer_number_status=VALID_CUSTOMER_NUMBER;
customer_number_length=strlen(possibly_valid_customer_number);
if (customer_number_length > MAXIMUM_CUSTOMER_NUMBER_LENGTH)
   {
   *customer_number_status=INVALID_CUSTOMER_NUMBER;
   }
else
   {
   for (customer_number_index=0; customer_number_index < customer_number_length; customer_number_index++)
      {
      if ((possibly_valid_customer_number[customer_number_index] < '0') ||
          (possibly_valid_customer_number[customer_number_index] > '9'))
         {
         *customer_number_status=INVALID_CUSTOMER_NUMBER;
         }
      }
   }
}
```

Figure 792.29: Example of identifier spellings containing lots of characters. Based on an example from Laitinen.[103]

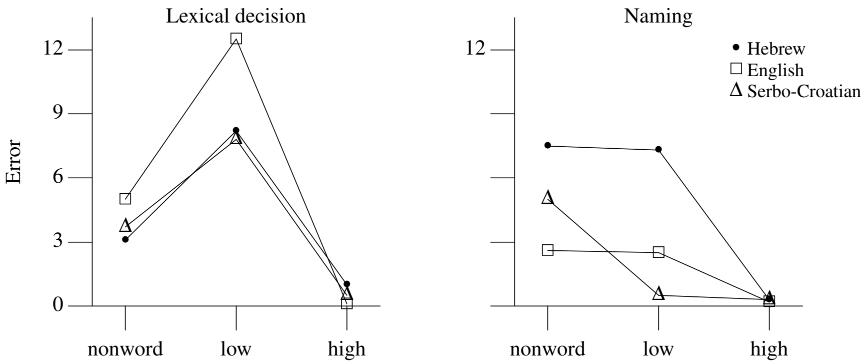


Figure 792.30: Error (as a percentage of responses) for naming and lexical decision tasks in Hebrew, English, and Serbo-Croatian using high/low frequency words and nonwords. Adapted from Frost, Katz, and Bentin.[71]

Table 792.20: WordNet 2.0 database statistics.

Part of Speech	Unique Strings	Synsets	Total Word-sense Pairs
Noun	114,648	79,689	141,690
Verb	11,306	13,508	24,632
Adjective	21,436	18,563	31,015
Adverb	4,669	3,664	5,808
Total	152,059	115,424	203,145

Table 792.21: The syllable most likely to be omitted in a word (indicated by the × symbol) based on the number of syllables (*syl*) and the position of the primary, (*pri*) stressed syllable. Adapted from Carter and Clopper.[30]

Syllables in Word and Primary Stress Position	Syllable(s) 1	Omitted 2	Most 3	Often 4
2syl–1pri		×	–	–
2syl–2pri	×		–	–
3syl–1pri		×	×	–
3syl–2pri	×			–
3syl–3pri		×	×	–
4syl–1pri		×		
4syl–2pri			×	×
4syl–3pri	×	×		×

Table 792.22: Five different applications (A–E) unabbreviated using `InName`, by five different people. Application C had many short names of the form `i`, `m`, `k`, and `r2`. Adapted from Laitinen.[104]

Application	A	B	C	D	E
Source lines	12,075	6,114	3,874	6,420	3,331
Total names	1,410	927	439	740	272
Already acceptable	5.6	3.1	8.7	9.3	11.0
Tool suggestion used	42.6	44.7	35.3	46.8	41.5
User suggestion used	39.6	29.3	15.0	30.7	43.8
Skipped or unknown names	12.2	22.9	41.0	13.2	3.7
User time (hours)	11	5	4	4	3

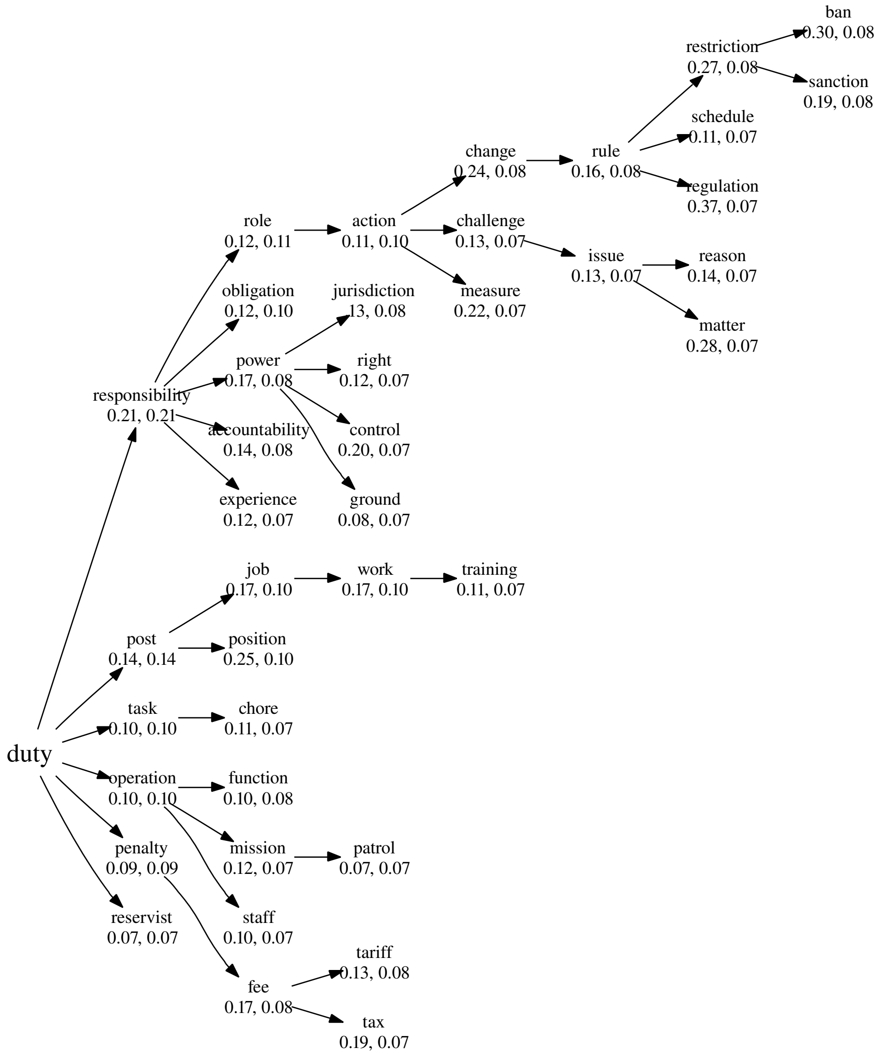


Figure 792.31: Semantic similarity tree for *duty*. The first value is the computed similarity of the word to its parent (in the tree), the second value its similarity to *duty*. Adapted from Lin.[112]

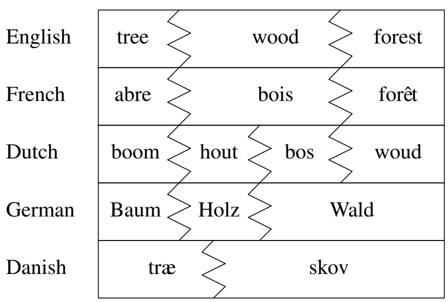


Figure 792.32: The relationship between words for tracts of trees in various languages. The interpretation given to words (boundary indicated by the zigzags) in one language may overlap that given in other languages. Adapted from DiMarco, Hirst, and Stede.[55]

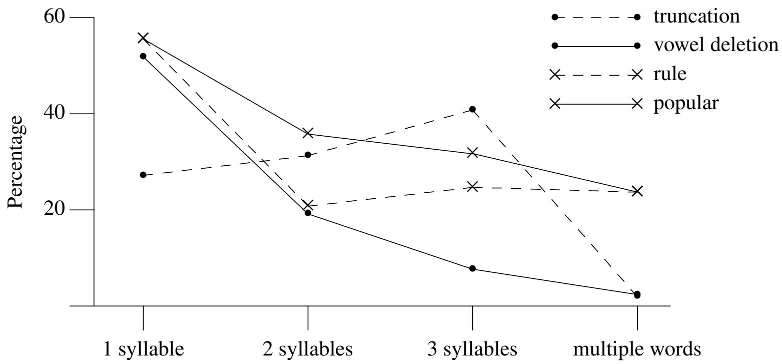


Figure 792.33: Percentage of abbreviations generated using each algorithm. The *rule* case was a set of syllable-based rules created by Streeter et al.; the *popular* case was the percentage occurrence of the most popular abbreviation. Based on Streeter, Ackroff, and Taylor.[160]

Table 792.23: Distribution of mistakes for each kind of text. Unparenthesized values are for subjects who made fewer than 2.5% mistakes, and parenthesized values for subjects who made 2.5% or more mistakes. Omission— failing to type a letter; response— hitting a key adjacent to the correct one; reading— mistakes were those letters that are confusable visually or acoustically; context — transpositions of adjacent letters and displacements of letters appearing within a range of three letters left or right of the mistake position; random— everything else. When a mistake could be assigned to more than one category, the category appearing nearer the top of the table was chosen. Adapted from Shaffer.[148]

Kind of mistake	Prose	Word	Syllable	First Order	Zero Order	Total
Omission	19 (21)	11 (23)	24 (36)	15 (46)	34 (82)	103 (208)
Response	19 (25)	31 (38)	27 (53)	32 (43)	108 (113)	217 (272)
Reading	3 (2)	2 (0)	8 (15)	14 (20)	20 (41)	47 (78)
Context	19 (27)	19 (17)	34 (30)	56 (51)	46 (40)	174 (165)
Random	3 (5)	2 (6)	4 (11)	13 (15)	22 (41)	44 (78)
Total	63 (80)	65 (84)	97 (145)	130 (175)	230 (317)	585 (801)

Table 792.24: Mean response time per letter (in milliseconds). Right half of the table shows mean response times for the same subjects with comparable passages in the first experiment. Adapted from Shaffer.[148]

	Syllable	Random		First Order	Zero Order
5-letter	246	326	Fixed	236	344
15-letter	292	373	Random	242	343

There is no specific limit on the maximum length of an identifier. 795

Usage

The distribution of identifier lengths is given in Figure 792.7.

The initial character shall not be a universal character name designating a digit. 797

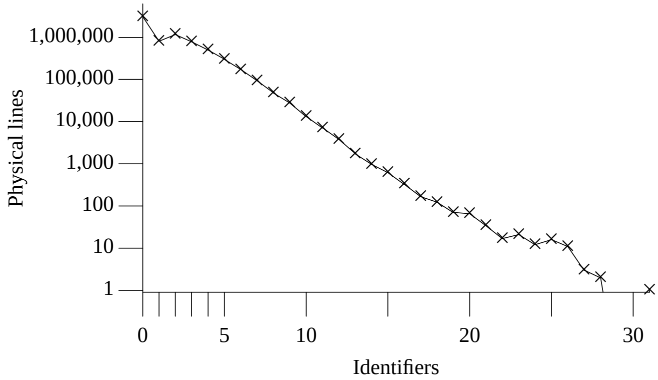


Figure 792.34: Number of physical lines containing a given number of identifiers. Based on the visible form of the `.c` files.

Table 797.1: The Unicode digit encodings.

Encoding Range	Language	Encoding Range	Language
0030–0039	ISO Latin-1	0BE7–0BEF	Tamil (has no zero)
0660–0669	Arabic–Indic	0C66–0C6F	Telugu
06F0–06F9	Eastern Arabic–Indic	0CE6–0CEF	Kannada
0966–096F	Devanagari	0D66–0D6F	Malayalam
09E6–09EF	Bengali	0E50–0E59	Thai
0A66–0A6F	Gurmukhi	0ED0–0ED9	Lao
0AE6–0AEF	Gujarati	FF10–FF19	Fullwidth
0B66–0B6F	Oriya digits		

806 The number of significant characters in an identifier is implementation-defined.

822 constant syntax

```
constant:
        integer-constant
        floating-constant
        enumeration-constant
        character-constant
```

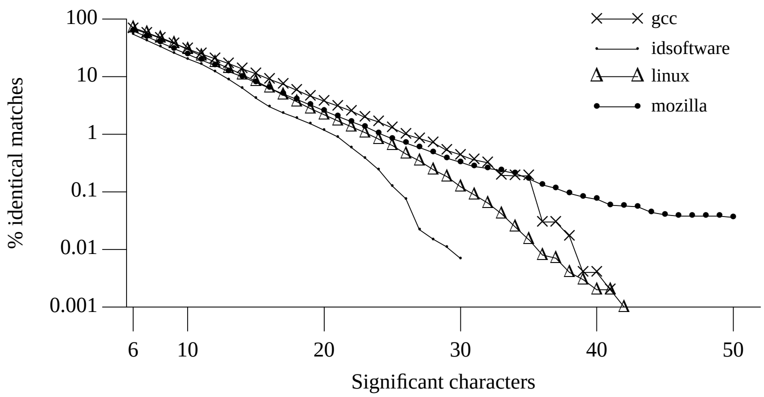


Figure 806.1: Occurrence of unique identifiers whose significant characters match those of a different identifier (as a percentage of all unique identifiers in a program), for various numbers of significant characters. Based on the visible form of the `.c` files.

Table 822.1: Occurrence of different kinds of constants (as a percentage of all tokens). Based on the visible form of the `.c` and `.h` files.

Kind of Constant	`.c` files	`.h` files
character-constant	0.16	0.06
integer-constant	6.70	20.79
floating-constant	0.02	0.20
string-literal	1.02	0.74

integer constant syntax

825

integer-constant:
    decimal-constant integer-suffix$_{opt}$
    octal-constant integer-suffix$_{opt}$
    hexadecimal-constant integer-suffix$_{opt}$
decimal-constant:
    nonzero-digit
    decimal-constant digit
octal-constant:
    0
    octal-constant octal-digit
hexadecimal-constant:
    hexadecimal-prefix hexadecimal-digit
    hexadecimal-constant hexadecimal-digit
hexadecimal-prefix: one of
    0x 0X
nonzero-digit: one of
    1 2 3 4 5 6 7 8 9
octal-digit: one of
    0 1 2 3 4 5 6 7
hexadecimal-digit: one of
    0 1 2 3 4 5 6 7 8 9
    a b c d e f
    A B C D E F
integer-suffix:
    unsigned-suffix long-suffix$_{opt}$
    unsigned-suffix long-long-suffix
    long-suffix unsigned-suffix$_{opt}$
    long-long-suffix unsigned-suffix$_{opt}$
unsigned-suffix: one of
    u U
long-suffix: one of
    l L
long-long-suffix: one of
    ll LL

Usage

integer constant usage

Having some forms of constant tokens (also see Figure 842.1) follow Benford's law[82] would not be surprising because the significant digits of a set of values created by randomly sampling from a variety of different distributions converges to a logarithmic distribution (i.e., Benford's law).[81] While the results for *decimal-constant* (see Figure 825.2) may appear to be a reasonable fit, applying a chi-squared test shows the fit to be remarkably poor (χ^2 = 132,398). The first nonzero digit of *hexadecimal-constant*s appears to be approximately evenly distributed.

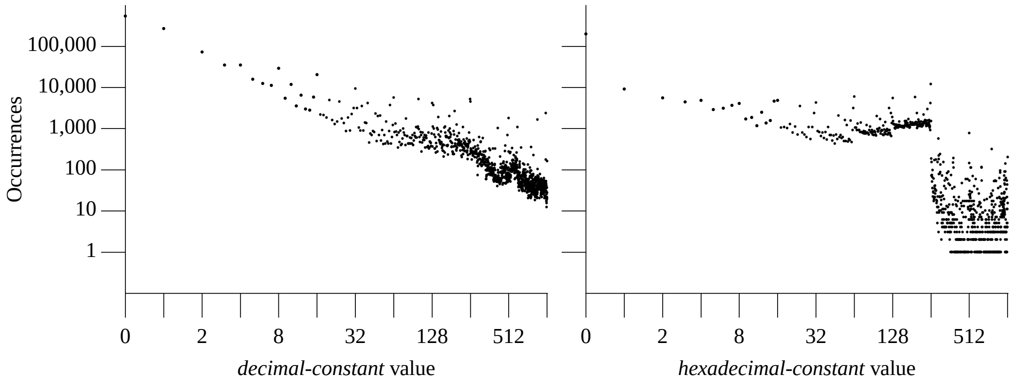


Figure 825.1: Number of integer constants having the lexical form of a *decimal-constant* (the literal 0 is also included in this set) and *hexadecimal-constant* that have a given value. Based on the visible form of the `.c` and `.h` files.

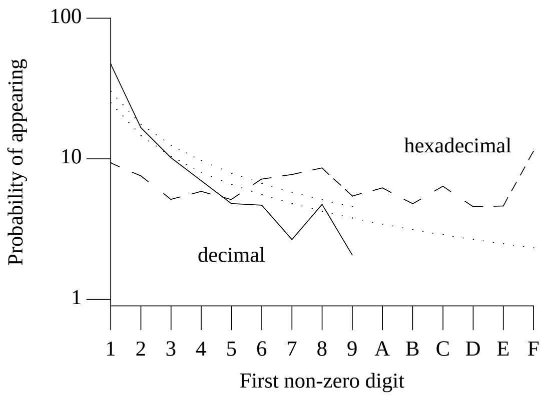


Figure 825.2: Probability of a *decimal-constant* or *hexadecimal-constant* starting with a particular digit; based on `.c` files. Dotted lines are the probabilities predicted by Benford's law (for values expressed in base 10 and base 16), i.e., $\log(1 + d^{-1})$, where d is the numeric value of the digit.

Table 825.1: Occurrence of various kinds of *integer-constant*s (as a percentage of all integer constants; note that zero is included in the *decimal-constant* count rather than the *octal-constant* count). Based on the visible form of the `.c` and `.h` files.

Kind of *integer-constant*	`.c` files	`.h` files
decimal-constant	64.1	17.8
hexadecimal-constant	35.8	82.1
octal-constant	0.1	0.2

Table 825.2: Occurrence of various *integer-suffix* sequences (as a percentage of all *integer-constants*). Based on the visible form of the `.c` and `.h` files.

Suffix Character Sequence	`.c` files	`h.` files	Suffix Character Sequence	`.c` files	`.h` files
none	99.6850	99.5997	Lu/lU	0.0005	0.0001
U/u	0.0298	0.0198	LL/lL/ll	0.0072	0.0022
L/l	0.1378	0.2096	ULL/uLl/ulL/Ull	0.0128	0.0061
U/uL/ul	0.1269	0.1625	LLU/lLu/LlU/llu	0.0000	0.0000

Table 825.3: Common token pairs involving *integer-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
, *integer-constant*	42.9	56.5	(*integer-constant*	2.8	3.4
integer-constant]	6.4	44.4	== *integer-constant*	25.5	2.0
integer-constant ,	58.2	44.2	**return** *integer-constant*	18.6	1.9
integer-constant ;	14.1	12.1	+ *integer-constant*	33.7	1.9
integer-constant)	14.2	11.7	& *integer-constant*	30.6	1.5
integer-constant #	1.4	9.1	identifier *integer-constant*	0.3	1.5
= *integer-constant*	19.6	9.0	- *integer-constant*	44.0	1.3
[*integer-constant*	39.3	5.6	< *integer-constant*	40.0	1.3
integer-constant }	1.2	4.4	{ *integer-constant*	4.2	1.2
-v *integer-constant*	69.0	4.1			

hexadecimal constant

A hexadecimal constant consists of the prefix **0x** or **0X** followed by a sequence of the decimal digits and the letters **a** (or **A**) through **f** (or **F**) with values 10 through 15 respectively. 830

Table 830.1: Occurrence of *hexadecimal-constant*s containing a given number of digits (as a percentage of all such constants). Based on the visible form of the `.c` files.

Digits	Occurrence	Digits	Occurrence	Digits	Occurrence	Digits	Occurrence
0	0.003	5	0.467	10	0.005	15	0.000
1	1.092	6	0.226	11	0.001	16	0.209
2	59.406	7	0.061	12	0.001		
3	1.157	8	2.912	13	0.000		
4	34.449	9	0.010	14	0.000		

integer constant type first in list

The type of an integer constant is the first of the corresponding list in which its value can be represented. 835

Table 835.1: Occurrence of *integer-constant*s having a particular type (as a percentage of all such constants; with the type denoted by any suffix taken into account) when using two possible representations of the type **int** (i.e., 16- and 32-bit). Based on the visible form of the `.c` and `.h` files.

Type	16-bit **int**	32-bit **int**
int	94.117	99.271
unsigned int	3.493	0.414
long	1.805	0.118
unsigned long	0.557	0.138
other-types	0.029	0.059

836

integer constant possible types

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	**int** **long int** **long long int**	**int** **unsigned int** **long int** **unsigned long int** **long long int** **unsigned long long int**
u or **U**	**unsigned int** **unsigned long int** **unsigned long long int**	**unsigned int** **unsigned long int** **unsigned long long int**
l or **L**	**long int** **long long int**	**long int** **unsigned long int** **long long int** **unsigned long long int**
Both **u** or **U** and **l** or **L**	**unsigned long int** **unsigned long long int**	**unsigned long int** **unsigned long long int**
ll or **LL**	**long long int**	**long long int** **unsigned long long int**
Both **u** or **U** and **ll** or **LL**	**unsigned long long int**	**unsigned long long int**

Suffix	Decimal Constant
none	**int** **long int** **unsigned long int**
l or **L**	**long int** **unsigned long int**

---

842 floating constant syntax

floating-constant:
  decimal-floating-constant
  hexadecimal-floating-constant
decimal-floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$
hexadecimal-floating-constant:
  hexadecimal-prefix hexadecimal-fractional-constant
    binary-exponent-part floating-suffix$_{opt}$
  hexadecimal-prefix hexadecimal-digit-sequence
    binary-exponent-part floating-suffix$_{opt}$
fractional-constant:
  digit-sequence$_{opt}$ **.** *digit-sequence*
  digit-sequence **.**
exponent-part:
  e *sign*$_{opt}$ *digit-sequence*
  E *sign*$_{opt}$ *digit-sequence*
sign: one of
  + -
digit-sequence:
  digit

```
                 digit-sequence digit
hexadecimal-fractional-constant:

                 hexadecimal-digit-sequence_opt .
                                  hexadecimal-digit-sequence
                 hexadecimal-digit-sequence .
binary-exponent-part:
                 p sign_opt digit-sequence
                 P sign_opt digit-sequence
hexadecimal-digit-sequence:
                 hexadecimal-digit
                 hexadecimal-digit-sequence hexadecimal-digit
floating-suffix: one of
                 f  l  F  L
```

Usage

exponent 334
integer 825
constant usage

Exponent usage information is given elsewhere. Also see elsewhere for a discussion of Benford's law and the first non-zero digit of constants (χ^2 = 1,680 is a very poor fit).

Table 842.1: Occurrence of various *floating-suffix*es (as a percentage of all such constants). Based on the visible form of the `.c` and `.h` files.

Suffix Character Sequence	.c files	.h files
none	98.3963	99.7554
F/f	1.4033	0.1896
L/l	0.2005	0.0550

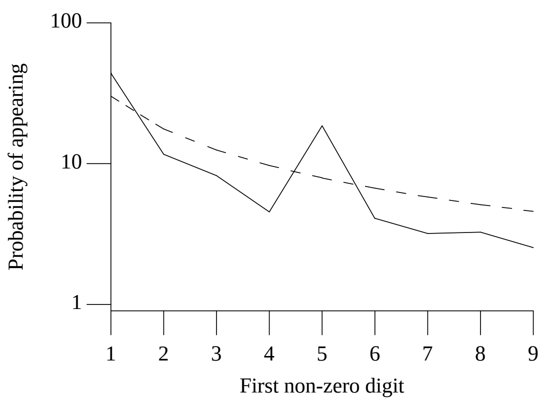


Figure 842.1: Probability of a *decimal-floating-constant* (i.e., not hexadecimal) starting with a particular digit. Based on the visible form of the `.c` files. Dotted line is the probability predicted by Benford's, i.e., $\log(1 + d^{-1})$, where d is the numeric value of the digit.

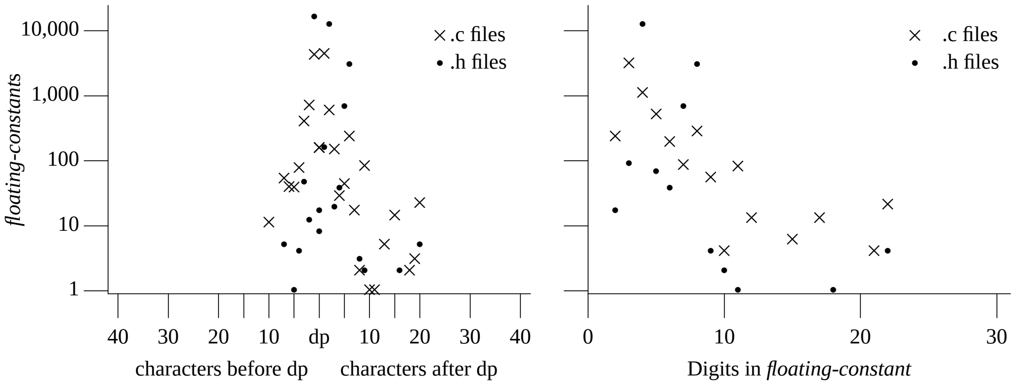


Figure 844.1: Number of *floating-constant*s, that do not contain an exponent part, containing a given number of digit sequences before and after the decimal point (dp), and the total number of digit in a *floating-constant*. Based on the visible form of the `.c` and `.h` files.

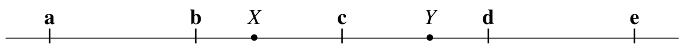


Figure 852.1: The nearest representable value to X is b, however, its value may also be rounded to a or c. In the case of Y, while d is the nearest representable value the result may be rounded to c or e.

Table 842.2: Common token pairs involving *floating-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
, *floating-constant*	0.0	20.4	*floating-constant* /	5.8	1.8
= *floating-constant*	0.1	15.7	*= *floating-constant*	6.3	1.6
* *floating-constant*	0.2	12.5	*floating-constant* *	6.8	0.1
(*floating-constant*	0.0	8.8	*floating-constant* ;	26.5	0.1
+ *floating-constant*	0.4	7.7	*floating-constant*)	25.9	0.1
-v *floating-constant*	0.3	6.7	*floating-constant* ,	25.8	0.1
/ *floating-constant*	2.0	6.4			

---

844 The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (`.`), followed by a digit sequence representing the fraction part. whole-number part fraction part

---

852 For decimal floating constants, and also for hexadecimal floating constants when **`FLT_RADIX`** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. floating constant representable value chosen

---

866 character constant syntax escape sequence syntax

```
character-constant:
        ' c-char-sequence '
        L' c-char-sequence '
c-char-sequence:
        c-char
        c-char-sequence c-char
c-char:
        any member of the source character set except
              the single-quote ', backslash \, or new-line character
        escape-sequence
```

```
escape-sequence:
                simple-escape-sequence
                octal-escape-sequence
                hexadecimal-escape-sequence
                universal-character-name
simple-escape-sequence: one of
                \'  \"  \?  \\
                \a  \b  \f  \n  \r  \t  \v
octal-escape-sequence:
                \ octal-digit
                \ octal-digit octal-digit
                \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
                \x hexadecimal-digit
                hexadecimal-escape-sequence hexadecimal-digit
```

Table 866.1: Occurrence of various kinds of *character-constant* (as a percentage of all such constants). Based on the visible form of the .c files.

Kind of *character-constant*	% of all *character-constant*s
not an escape sequence	76.1
simple-escape-sequence	8.8
octal-escape-sequence	15.1
hexadecimal-escape-sequence	0.0
universal-character-name	0.0

Table 866.2: Occurrence of *escape-sequence*s within *character-constant*s and *string-literal*s (as a percentage of *escape-sequence*s for that kind of token). Based on the visible form of the .c files.

Escape Sequence	% of *character-constant* Escape Sequences	% of *string-literal* escape sequences	Escape sequence	% of *character-constant* Escape Sequences	% of *string-literal* Escape Sequences
\n	18.10	79.15	\b	0.66	0.04
\t	3.90	11.62	\'	3.24	0.02
\"	1.29	3.08	\%	0.00	0.02
\0	52.70	2.06	\v	0.31	0.01
\x	0.12	1.10	\p	0.00	0.01
\2	2.73	1.01	\f	0.44	0.01
\\	5.70	0.61	\?	0.01	0.01
\r	3.01	0.46	\e	0.00	0.00
\3	4.95	0.42	\a	0.11	0.00
\1	2.72	0.35			

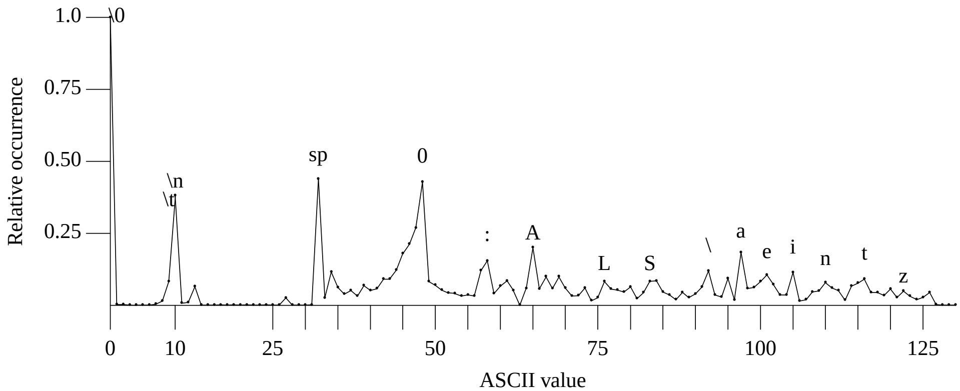


Figure 884.1: Relative frequency of occurrence of characters in an integer *character-constant* (as a fraction of the most common character, the null character). Based on the visible form of the `.c` files.

Table 866.3: Common token pairs involving *character-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
`==` *character-constant*	7.1	22.8	*character-constant* `||`	4.2	4.2
`,` *character-constant*	0.3	18.1	*character-constant* `&&`	5.3	3.3
case *character-constant*	8.5	16.7	`<=` *character-constant*	7.1	1.7
`=` *character-constant*	0.8	14.2	`>=` *character-constant*	3.6	1.5
`!=` *character-constant*	5.3	8.4	*character-constant* `)`	33.0	0.7
`(` *character-constant*	0.1	6.1	*character-constant* `,`	17.6	0.3
character-constant `:`	16.7	6.0	*character-constant* `;`	16.6	0.3

character constant value

884 The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer.

Table 884.1: Occurrence of a *character-constant* appearing as one of the operands of various kinds of binary operators (as a percentage of all such constants; includes escape sequences). Based on the visible form of the `.c` files. See Table 866.3 for more detailed information.

Operator	%
Arithmetic operators	4.5
Bit operators	0.5
Equality operators	31.3
Relational operators	4.1

character constant more than one character

885 The value of an integer character constant containing more than one character (e.g., **`'ab'`**), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined.

Table 885.1: Number of *character-constant*s containing a given number of characters. Based on the visible form of the `.c` files.

Number of Characters	Occurrences	Number of Characters	Occurrences
0	27	4	21
1	50,590	5	4
2	0	6	4
3	8	7	0

string literal syntax

895

```
string-literal:
        " s-char-sequence_opt "
        L" s-char-sequence_opt "
s-char-sequence:
        s-char
        s-char-sequence s-char
s-char:
        any member of the source character set except
                the double-quote ", backslash \\, or new-line character
        escape-sequence
```

Usage

Usage of escape sequences in string literal and string lengths is given elsewhere (see Table 866.2 and Figure 293.1).

In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and wide string literal tokens are concatenated into a single multibyte character sequence. 899

Usage

line splicing

In the visible form of the `.c` files 4.9% (`.h` 15.6%) of all string literals are concatenated (i.e., immediately adjacent to another string literal) and 1.4% (`.h` 10.7%) occupied more than one source line (i.e., line splicing occurred).

The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined. 907

Usage

In the visible form of the `.c` files 2.1% (`.h` 2.9%) of characters in string literals are not in the basic execution character set (the value of escape sequences were compared using the values of the Ascii character set).

string literal distinct array

It is unspecified whether these arrays are distinct provided their elements have the appropriate values. 908

Table 908.1: Number of *string-literal*s (the empty *string-literal*, i.e., `""`, was not counted). Based on the visible form of the `.c` and `.h` files. Although many of the program source trees contain more than one program, they were treated as a single entity. A consequence of this is that the number of unique matches represents a lower bound; having a smaller number of string literals is likely to reduce the probability of matches occurring.

	gcc	idsoftware	linux	netscape	openafs	openMotif	postgresql	Total
Number of strings	38,063	21,811	177,224	30,358	30,574	11,285	16,387	325,702
Bytes in strings	656,366	324,667	4,050,258	512,766	737,015	288,018	298,888	6,867,978
Number of unique strings	18,602	9,148	114,170	17,192	18,483	7,401	7,930	187,549
Bytes in unique strings	434,028	170,170	3,189,466	378,917	562,555	240,811	219,690	5,159,385

912

punctuator syntax

punctuator: one of

```
[  ]  (  )  {  }  .  ->
++  --  &  *  +  -  ~  !
/  %  <<  >>  <  >  <=  >=  ==  !=  ^  | &&  ||
?  :  ;  ...
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #  ##
<:  :>  <%  %>  %:  %:%:
```

Table 912.1: Commonly used terms for punctuators and operators.

Punctuator/ Operator	Term	Punctuator/ Operator	Term
[ ]	left square bracket or opening square bracket or bracket	^	circumflex or xor or exclusive *or*
( )	left round bracket or opening round bracket or bracket or parenthesis	|	vertical bar or bitwise *or* or *or*
{ }	left curly bracket or opening curly bracket or bracket or brace	&&	and and or logical and
.	dot or period or full stop or dot selection	||	logical *or* or *or*
->	indirect or indirect selection	?	question mark
*	times or star or dereference or asterisk	:	colon
+	plus	;	semicolon
-	minus or subtract	...	dot dot dot or ellipsis
~	tilde or bitwise not	=	equal or assign
!	exclamation or shriek	*=	times equal
++	plus plus	/=	divide equal
--	minus minus	%=	percent equal or remainder equal
&	and or address of or ampersand or bitwise-and	+=	plus equal
/	slash or divide or solidus	-=	minus equal
%	remainder or percent	<<=	left-shift equal
<<	left-shift	>>=	right-shift equal
>>	right-shift	&=	and equal
<	less than	^=	xor equal or exclusive or equal
>	greater than	|=	or equal
<=	less than or equal	,	comma
>=	greater than or equal	#	hash or sharp or pound
==	equal	##	hash hash or sharp sharp or pound pound
!=	not equal	<: :> <% %> %: %:%:	no commonly used terms

Table 912.2: Occurrence of *punctuator* tokens (as a percentage of all tokens; multiply by 1.88 to express occurrence as a percentage of all punctuator tokens). Based on the visible form of the .c and .h files.

Punctuator	% of Tokens	Punctuator	% of Tokens	Punctuator	% of Tokens	Punctuator	% of Tokens
,	8.82	==	0.53	||	0.16	-=	0.03
)	8.09	:	0.46	+=	0.11	++v	0.02
(	8.09	-v	0.40	>	0.11	%	0.02
;	7.80	*p	0.40	<<	0.09	--v	0.01
=	3.08	+	0.38	?:	0.08	...	0.01
->	3.00	*v	0.34	?	0.08	>>=	0.01
}	1.87	&	0.32	|=	0.08	^	0.01
{	1.87	!	0.31	>=	0.07	+v	0.00
.	1.26	v++	0.27	/	0.06	%=	0.00
*	1.10	&&	0.26	>>	0.06	##	0.00
#	1.00	!=	0.26	~	0.05	*=	0.00
]	0.96	<	0.22	v--	0.04	/=	0.00
[	0.96	-	0.19	&=	0.04	<<=	0.00
&v	0.58	|	0.17	<=	0.04	^=	0.00

digraphs

In all aspects of the language, the six tokens[67)] 916

```
<:  :>  <%  %>  %:  %:%:
```

behave, respectively, the same as the six tokens

```
[  ]  {  }  #  ##
```

except for their spelling.68)

Usage

The visible form of the `.c` files contained zero digraphs.

---

918 header name syntax

```
header-name:
                < h-char-sequence >
                " q-char-sequence "
h-char-sequence:
                h-char
                h-char-sequence h-char
h-char:
                any member of the source character set except
                                the new-line character and >
q-char-sequence:
                q-char
                q-char-sequence q-char
q-char:
                any member of the source character set except
                                the new-line character and "
```

Usage

Header name usage information is given elsewhere. 1896 source file inclusion

---

933 DR324) For an example of a header name preprocessing token used in a **#pragma** directive, see Subclause 6.10.9. footnote DR324

Usage

While over 30% of the characters in this book's benchmark programs (see Table 770.3) are contained within comments, they only represent around 2% of the tokens. A study by Fluri et al[68] of the releases of three large Java programs over a 6 year period (on average) found three different patterns in the ratio of number of comment lines to number of non-comment lines for each program.

A study of comments in C++ source by Etzkorn[65] found that 57% contained English sentences (that could be automatically parsed by the tool used).

Table 933.1: Common formats of nonsentence style comments. Adapted from Etzkorn, Bowen, and Davis.[65]

Style of Comment	Example
Item name— Definition	MaxLength— Maximum CFG Depth.
Definition	Maximum CFG Depth.
Unattached prepositional phrase	To support scrolling text.
Value definitions	0 = not selected, 1 = is selected.
Mathematical formulas	Can be Boolean expressions...

Table 933.2: Breakdown of comments containing parsable sentences. Adapted from Etzkorn, Bowen, and Davis.[65]

Percentage	Style of Sentence	Example
51	Operational description	This routine reads the data. Then it opens the file.
44	Definition	General Matrix— rectangular matrix class.
2	Description of definition	This defines a NIL value for a list.
3	Instructions to reader	See the header at the top of the file.

Table 933.3: Common formats of sentence-style comments. Adapted from Etzkorn, Bowen, and Davis.[65]

Part of Speech	Percentage	Example
Present Tense	75	
Indicative mood, active voice		This routine reads the data.
Indicative mood, active voice, missing subject		Reads the data.
Imperative mood, active voice		Read the data.
Indicative mood, passive voice		This is done by reading the data.
Indicative mood, passive voice, missing subject		Is done by reading the data.
Past Tense	4	
Indicative mood, either active or passive voice, occasional missing subject		This routine opened the file. or Opened the file.
Future Tense	4	
Indicative mood, either active or passive voice, occasional missing subject		This routine will open the file. or Will open the file.
Other	15	

comment /*

Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. 934

Table 934.1: Four types of questions.

Statement Relative to Fact	Example
true-affirmative (TA)	star is above plus: ∗ +
false-affirmative (FA)	plus is above star: + ∗
false-negative (FN)	star isn't above plus: ∗ +
true-negative (TN)	plus isn't above star: ∗ +

Table 934.2: Occurrence of kinds of comments (as a percentage of all comments; last row as a percentage of all new-line characters). Based on the visible form of the `.c` and `.h` files.

Kind of Comment	`.c` files	`.h` files
/* comment */	91.0	90.1
// comment	9.0	9.9
/* on one line */	70.3	79.1
new-lines in /* comments	12.3	17.5

expressions

An *expression* is a sequence of operators and operands that specifies computation of a value, or that 940

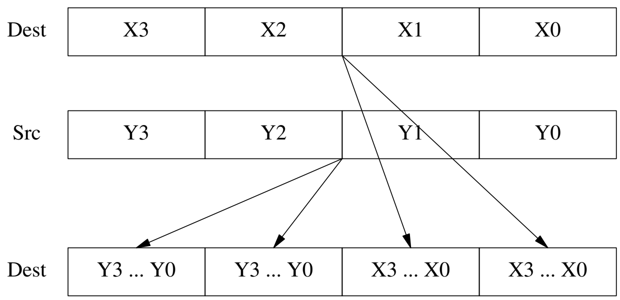


Figure 940.1: The SHUFPS (shuffle packed single-precision floating-point values) instruction, supported by the Intel Pentium processor,[87] places any two of the four packed floating-point values from the destination operand into the two low-order doublewords of the destination operand, and places any two of the four packed floating-point values from the source operand into the two high-order doublewords of the destination operand. By using the same register for the source and destination operands, the SHUFPS instruction can shuffle four single-precision floating-point values into any order.

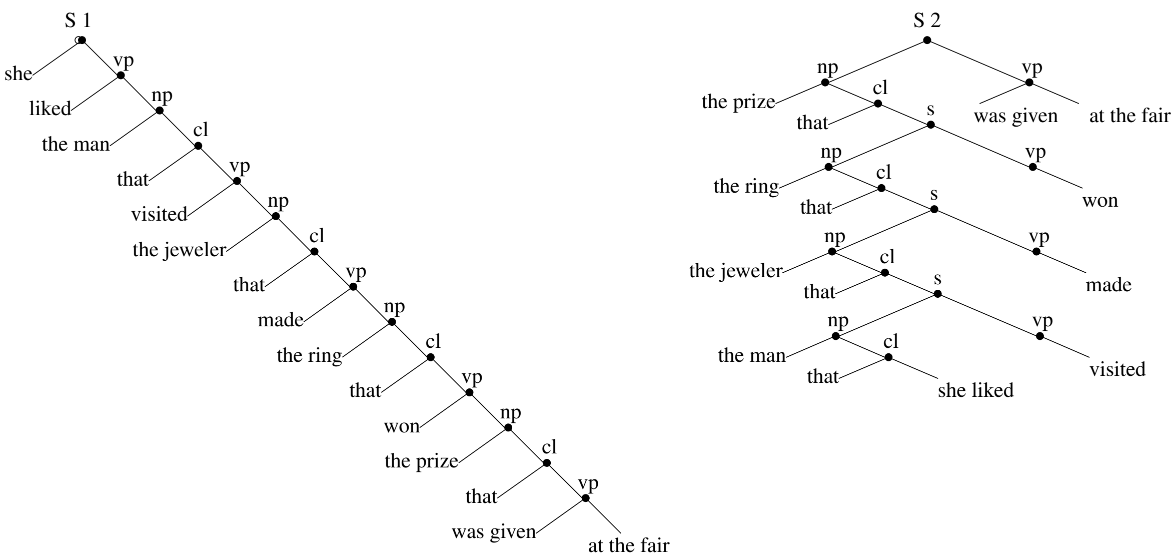


Figure 940.2: Parse tree of a sentence with no embedding (S 1) and a sentence with four degrees of embedding (S 2). Adapted from Miller and Isard.[123]

designates an object or a function, or that generates side effects, or that performs a combination thereof.

Table 940.1: Occurrence of a token as the last token on a physical line (as a percentage of all occurrences of that token and as a percentage of all lines). Based on the visible form of the `.c` files.

Token	% Occurrence of Token	% Last Token on Line	Token	% Occurrence of Token	% Last Token on Line
;	92.2	36.0	**#else**	89.1	0.2
* ... *\	97.9	8.4	**int**	5.3	0.2
)	20.6	8.3	||	23.7	0.2
{	86.7	8.1	|	12.3	0.1
}	78.9	7.4	**+**	3.8	0.1
,	13.9	6.1	**?:**	7.3	0.0
:	74.3	1.7	**?**	7.1	0.0
header-name	97.7	1.5	**do**	21.3	0.0
\\	100.0	0.9	**#error**	25.1	0.0
#endif	81.9	0.8	**:b**	7.2	0.0
else	42.2	0.7	**double**	3.1	0.0
string-literal	8.0	0.4	**^**	3.1	0.0
void	18.2	0.4	**union**	6.2	0.0
&&	17.8	0.2			

Usage

partial redundancy elimination

A study by Bodík, Gupta, and Soffa[19] found that 13.9% of the expressions in SPEC95 were partially redundant, that is, their evaluation is not necessary under some conditions.

full expression 1712

See Table 1713.1 for information on occurrences of full expressions, and Table 770.2 for visual spacing between binary operators and their operands.

Table 940.2: Occurrence of a token as the first token on a physical line (as a percentage of all occurrences of that token and as a percentage of all lines). /* *new-line* */ denotes a comment containing one or more new-line characters, while /* ... */ denotes that form of comment on a single line. Based on the visible form of the `.c` files.

Token	% First Token on Line	% Occurrence of Token	Token	% First Token on Line	% Occurrence of Token
default	0.2	99.9	**volatile**	0.0	50.0
#	5.0	99.9	**int**	1.8	47.0
typedef	0.1	99.8	**unsigned**	0.7	46.8
static	2.1	99.8	**struct**	1.1	38.9
for	0.8	99.7	**const**	0.1	35.5
extern	0.2	99.6	**char**	0.5	30.5
switch	0.3	99.4	**void**	0.6	28.7
case	1.6	97.8	***v**	0.5	28.7
* new-line *\	13.7	97.7	**++v**	0.0	27.8
register	0.2	95.0	**signed**	0.0	27.2
return	3.3	94.5	**&&**	0.3	21.2
goto	0.4	94.1	identifier	31.1	20.8
if	6.9	93.6	||	0.2	18.4
break	1.2	91.8	**--v**	0.0	17.9
continue	0.2	91.3	**short**	0.0	16.0
}	8.3	88.3	**#error**	0.0	15.6
do	0.1	87.3	*string-literal*	0.6	12.4
while	0.4	85.2	**sizeof**	0.1	11.3
enum	0.1	73.7	**long**	0.1	10.1
\\	0.6	70.8	*integer-constant*	2.2	6.6
else	1.1	70.2	**?**	0.0	5.6
union	0.0	63.3	**&v**	0.1	5.2
* ... *\	5.4	62.6	**-v**	0.1	5.0
{	5.1	54.9	**?:**	0.0	5.0
float	0.0	54.0	|	0.0	4.2
double	0.0	53.6	*floating-constant*	0.0	4.1

Table 940.3: Breakdown of invariance by instruction types. These categories include integer loads (*ILd*), floating-point loads (*FLd*), load address calculations (*LdA*), stores (*St*), integer multiplication (*IMul*), floating-point multiplication (*FMul*), floating-point division (*FDiv*), all other integer arithmetic (*IArth*), all other floating-point arithmetic (*FArith*), compare (*Cmp*), shift (*Shft*), conditional moves (*CMov*), and all other floating-point operations (*FOps*). The first number shown is the percent invariance of the topmost value for a class type, while the number in parenthesis is the dynamic execution frequency of that type. Results are not shown for instruction types that do not write a register (e.g., branches). Adapted from Calder, Feller, and Eustace.[25]

Program	ILd	FLd	LdA	St	IMul	FMul	FDiv	IArth	FArth	Cmp	Shft	CMov	FOps
compress	44(27)	0(0)	88(2)	16(9)	15(0)	0(0)	0(0)	11(36)	0(0)	92(2)	14(9)	0(0)	0(0)
gcc	46(24)	83(0)	59(9)	48(11)	40(0)	30(0)	31(0)	46(28)	0(0)	87(3)	54(7)	51(1)	95(0)
go	36(30)	100(0)	71(13)	35(8)	18(0)	100(0)	0(0)	29(31)	0(0)	73(4)	42(0)	52(1)	100(0)
ijpeg	19(18)	73(0)	9(11)	20(5)	10(1)	68(0)	0(0)	15(37)	0(0)	96(2)	17(21)	15(0)	98(0)
li	40(30)	100(0)	27(8)	42(15)	30(0)	13(0)	0(0)	56(22)	0(0)	93(2)	79(3)	60(0)	100(0)
perl	70(24)	54(3)	81(7)	59(15)	2(0)	50(0)	19(0)	65(22)	34(0)	87(4)	69(6)	28(1)	51(1)
m88ksim	76(22)	59(0)	68(8)	79(11)	33(0)	53(0)	66(0)	64(28)	100(0)	91(5)	66(6)	65(0)	100(0)
vortex	61(29)	99(0)	46(6)	65(14)	9(0)	4(0)	0(0)	70(31)	0(0)	98(2)	40(3)	20(0)	100(0)

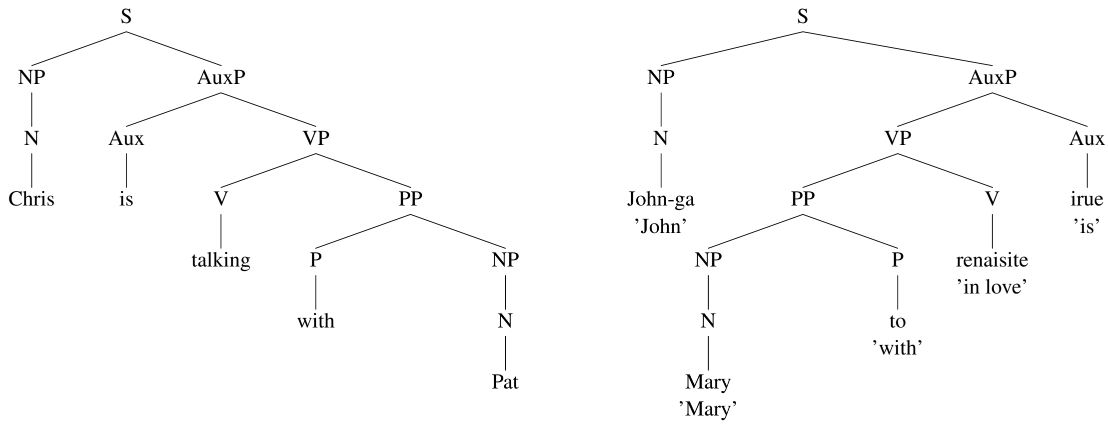


Figure 943.1: English ("Chris is talking with Pat") and Japanese ("John-ga Mary to renaisite irue") language phrase structure for sentences of similar complexity and structure. While the Japanese structure may seem back-to-front to English speakers, it appears perfectly natural to native speakers of Japanese. Adapted from Baker.[13]

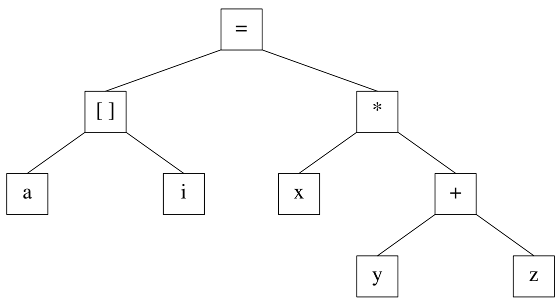


Figure 944.1: A simplified form of the kind of tree structure that is likely to be built by a translator for the expression `a[i]=x*(y+z)`.

Table 940.4: Number of objects defined (in a variety of small multimedia and scientific programs) to have types represented using a given number of bits (i.e., mostly 32-bit **`int`**) and number of objects having a maximum bit-width usage (i.e., number of bits required to represent any of the values stored in the object; rounded up to the nearest byte boundary). Adapted from Stephenson,[156] whose analysis was performed by static analysis of the source.

Bits	Objects Defined	Objects Requiring Specified Bits
1	0	203
8	7	134
16	27	108
32	686	275

expression grouping

The grouping of operators and operands is indicated by the syntax.72) 943

operator precedence

expression order of evaluation

Except as specified later (for the function-call **`()`**, **`&&`**, **`||`**, **`?:`**, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified. 944

identifier is primary expression if

An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).77) 976

Usage

A study by Yang and Gupta[174] found, for the SPEC95 programs, on average eight different values occupied 48% of all allocated storage locations throughout the execution of the programs. They called this behavior *frequent value locality*. The eight different values varied between programs and contained small values (zero was often the most frequently occurring value) and very large values (often program-specific addresses of objects and string literals).

Table 976.1: Dynamic percentage of load instructions from different *classes*. The *Class* column is a three-letter acronym: the first letter represents the region of storage (Stack, Heap, or Global), the second denotes the kind of reference (Array, Member, or Scalar), and the third indicates the type of the reference (Pointer or Nonpointer). For instance, *HFP* is a load of pointer-typed member from a heap-allocated object. There are two kinds of loads generated as a result of internal translator housekeeping: *RA* is a load of the return address from a function-call, and any register values saved to memory prior to the call also need to be reloaded when the call returns, *CS* callee-saved registers The figures were obtained by instrumenting the source prior to translation. As such they provide a count of loads that would be made by the abstract machine (apart from *RA* and *CS*). The number of loads performed by the machine code generated by translators is likely to be optimized (evaluation of constructs moved out of loops and register contents reused) and resulting in fewer loads. Whether these optimizations will change the distribution of loads in different classes is not known. Adapted from Burtscher, Diwan and Hauswirth.[24]

Class	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex	bzip	gzip	mcf	Mean
SSN	–	1.28	3.50	0.42	4.40	12.10	6.23	7.26	0.12	0.15	0.15	2.97
SAN	–	0.63	1.01	16.61	–	0.45	2.58	–	12.73	0.01	–	2.84
SMN	–	0.67	–	3.62	–	0.30	–	2.60	–	–	–	0.60
SSP	–	0.37	–	0.17	1.40	–	–	0.33	–	0.02	–	0.19
SAP	–	0.25	–	0.17	–	–	–	–	–	–	–	0.04
SMP	–	0.29	–	0.25	0.01	0.24	2.15	0.05	–	–	–	0.25
HSN	–	0.88	–	14.75	3.51	–	8.07	7.32	0.27	0.01	0.20	2.92
HAN	–	7.39	–	48.55	–	–	4.30	5.39	31.83	–	2.75	8.35
HMN	–	16.37	–	0.76	8.80	6.11	8.42	0.85	–	3.54	27.35	6.02
HSP	–	0.33	–	–	1.82	–	20.01	7.64	–	–	–	2.48
HAP	–	9.42	–	1.33	0.56	–	3.02	4.97	–	–	0.88	1.68
HMP	–	1.82	–	0.11	24.44	0.57	6.29	0.16	–	0.01	17.47	4.24
GSN	43.46	11.10	14.23	0.45	12.76	17.49	16.81	27.79	43.71	43.75	3.12	19.56
GAN	19.27	6.51	52.03	3.00	–	21.86	–	0.03	3.63	26.24	–	11.05
GMN	–	0.81	–	0.41	–	10.96	–	0.16	–	–	2.79	1.26
GSP	–	0.68	–	0.04	–	–	–	–	–	–	0.48	0.10
GAP	–	2.17	–	–	–	0.86	–	0.60	0.41	–	4.72	0.73
GMP	–	0.77	–	0.20	–	0.07	–	–	–	–	0.26	0.11
RA	7.65	5.16	3.68	0.91	8.84	4.58	4.11	4.60	0.76	2.52	7.29	4.17
CS	29.62	33.10	25.55	8.27	33.46	24.40	18.01	30.24	6.54	23.75	32.55	22.12

Table 976.2: Occurrence of load instructions (as a percentage of all instructions executed on HP–was DEC– Alpha). The column headed *Leaf* lists percentage of calls to leaf functions, *NonLeaf* is for calls to nonleaf functions. Adapted from Calder, Grunwald, and Zorn.[27]

Program	Mean	Leaf	NonLeaf	Program	Mean	Leaf	Non-Leaf
burg	21.7	12.9	26.7	eqntott	12.8	11.8	20.2
ditroff	30.3	18.6	32.9	espresso	21.6	20.1	22.9
tex	30.7	19.6	31.3	gcc	23.9	16.7	24.6
xfig	23.5	15.6	25.8	li	28.1	44.1	26.3
xtex	23.2	16.1	28.2	sc	21.2	15.3	22.8
compress	26.4	0.1	26.5	Mean	23.9	17.3	26.2

Table 976.3: Comparison of percentage of load instructions executed on Alpha and MIPS. Adapted from Calder, Grunwald, and Zorn.[27]

Program	MIPS	Alpha	Program	MIPS	Alpha
compress	17.3	26.4	li	21.8	28.1
eqntott	14.6	12.8	sc	19.2	21.2
espresso	17.9	21.6	Program mean	18.2	22.3
gcc	18.7	23.9			

A constant is a primary expression. 977

Usage

integer constant syntax 825

Usage information on the distribution of all constant values occurring in the source is given elsewhere.

A string literal is a primary expression. 979

Usage

string literal syntax 895

Usage information on string literals is given elsewhere.

---

parenthesized expression

A parenthesized expression is a primary expression. 981

Usage

parenthesized expression nesting levels 281

Usage information on parentheses usage is given elsewhere.

---

postfix-expression syntax

985

postfix-expression:
  primary-expression
  postfix-expression **[** *expression* **]**
  postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
  postfix-expression **.** *identifier*
  postfix-expression **->** *identifier*
  postfix-expression **++**
  postfix-expression **--**
  (*type-name* **) {** *initializer-list* **}**
  (*type-name* **) {** *initializer-list* **, }**
argument-expression-list:
  assignment-expression
  argument-expression-list **,** *assignment-expression*

Table 985.1: Occurrence of postfix operators having particular operand types (as a percentage of all occurrences of each operator, with `[` denoting array subscripting). Based on the translated form of this book's benchmark programs.

Operator	Type	%	Operator	Type	%
v++	int	54.0	[	unsigned char	5.1
v--	int	52.5	[	other-types	4.7
[	*	38.0	[	int	4.1
v++	*	25.7	v++	unsigned long	3.1
v--	long	15.9	v--	unsigned short	2.7
[	struct	14.5	v--	unsigned char	2.6
v++	unsigned int	13.3	[	const char	2.4
[	float	12.0	[	unsigned long	1.2
v--	unsigned int	11.5	v++	long	1.1
[	union	10.2	[	unsigned int	1.1
v--	*	7.1	v++	unsigned short	1.0
[	char	6.8	v++	unsigned char	1.0
v--	unsigned long	6.1	v--	short	1.0

Table 985.2: Common token pairs involving `.`, `->`, `++`, or `--` (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier ->	9.8	97.5	v++)	41.4	1.4
identifier v++	0.9	96.9	v++ ;	39.9	1.4
identifier v--	0.1	96.1	v++]	4.6	1.3
identifier .	3.6	83.8	v++ =	7.6	0.7
] .	20.3	15.4	v-- ;	58.4	0.3
-> identifier	100.0	10.1	v--)	29.1	0.1
. identifier	100.0	4.2			

988 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object.

Time step	Operation
t=1	A[0] = B[1]
t=2	C[1] = A[1]
t=3	A[1] = B[2]
t=4	C[2] = A[2]

Time step	Thread 1	Thread 2
t=1	A[0] = B[1]	A[1] = B[2]
t=2	C[1] = A[1]	C[2] = A[2]

Time step	Operation
t=1	B[1] = A[0]
t=2	A[1] = C[1]
t=3	B[2] = A[1]
t=4	A[2] = C[2]

Time step	Thread 1	Thread 2
t=1	B[1] = A[0]	B[2] = A[1]
t=2	A[1] = C[1]	A[2] = C[2]

Time step	Operation
t=1	A[0] = B[1]
t=2	A[1] = C[1]
t=3	A[1] = B[2]
t=4	A[2] = C[2]

Time step	Thread 1	Thread 2
t=1	A[0] = B[1]	A[1] = B[2]
t=2	A[1] = C[1]	A[2] = C[2]

Successive subscript operators designate an element of a multidimensional array object. 991

Table 991.1: Occurrence of object declarations having an array type with the given number of dimensions (as a percentage of all array types in the given scope; with local scope separated into parameters and everything else). Based on the translated form of this book's benchmark programs.

Dimensions Scope	Parameters File Scope	Local	non-parameter
1	100.0	97.9	91.9
2	0.0	2.0	7.5
3	0.0	0.1	0.6

---

array row-major storage order

It follows from this that arrays are stored in row-major order (last subscript varies fastest). 994

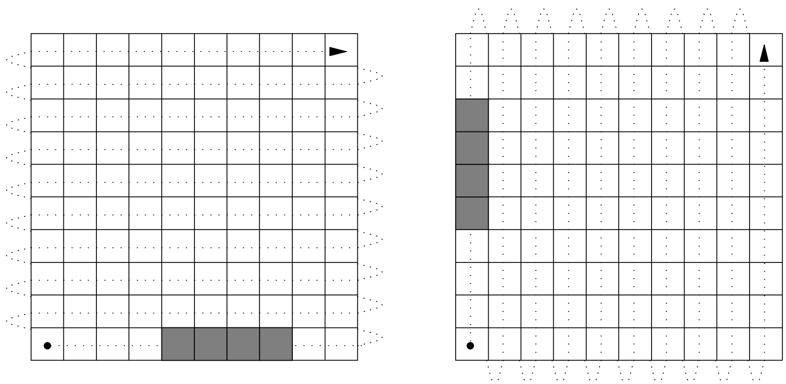


Figure 994.1: Row (left) and column (right) major order. The dotted line indicates successively increasing addresses for the two kinds of storage layouts, with the gray boxes denoting the same sequence of index values.

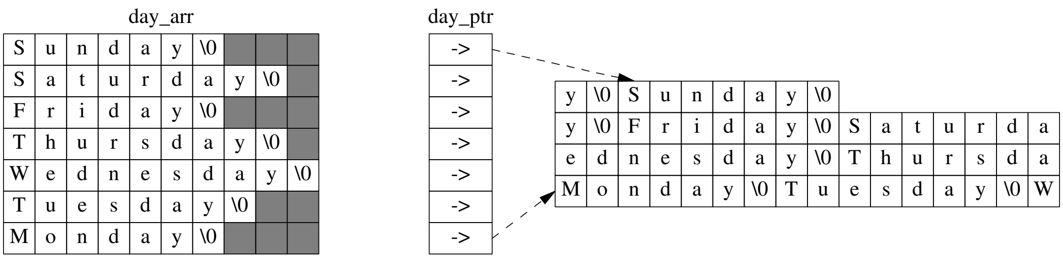


Figure 994.2: Difference in storage layout between an array of array of characters (left) and array of pointer to characters (right; not all pointers shown and the relative storage locations of the strings is only one of many that are possible).

Table 994.1: Cache hit-rate for sequentially accessing, in row-major order, a two-dimensional array stored using various layout methods. If the same array is accessed in column-major order the figures given in the Row-major and Column-major columns are swapped and the Morton layout figure remains unchanged. These figures ignore the impact that accessing other objects might have on cache behavior, and so denote the best hit-rate that can be achieved. Based on Thiyagalingam et al.[164]

Cache size	Row-major	Morton	Column-major
32 byte cache line	75%	50%	0%
128 cache byte	93.75%	75%	0%
8K byte cache page	99.9%	96.875%	0%

1000 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. operator ()

Usage

How frequent are function calls? The machine code instructions used to call a function may be generated by translators for reasons other than a function call in the source code. Some operators may be implemented via a call to an internal system library routine; for instance, floating-point operations on processors that do not support such operators in hardware. Such usage will vary between processors (see Figure 0.5).

	0	1	2	3	4	5	6	7
7	44	45	46	47	60	61	62	63
6	40	41	42	43	56	57	58	59
5	36	37	38	39	52	53	54	55
4	32	33	34	35	48	49	50	51
3	12	13	14	15	28	29	30	31
2	8	9	10	11	24	25	26	27
1	4	5	6	7	20	21	22	23
0	0	1	2	3	16	17	18	19

	0	1	2	3	4	5	6	7
7	42	43	46	47	58	59	62	63
6	40	41	44	45	56	57	60	61
5	34	35	38	39	50	51	54	55
4	32	33	36	37	48	49	52	53
3	10	11	14	15	26	27	30	31
2	8	9	12	13	24	25	28	29
1	2	3	6	7	18	19	22	23
0	0	1	4	5	16	17	20	21

Figure 994.3: Two possible element layouts of an $8 * 8$ array; Blocked row-major layout (left) and Morton element layout (right). Factors such as efficiency of array index calculation, whether array size can be made a power of two, or array shape (e.g., non-square) drive layout selection.[163]

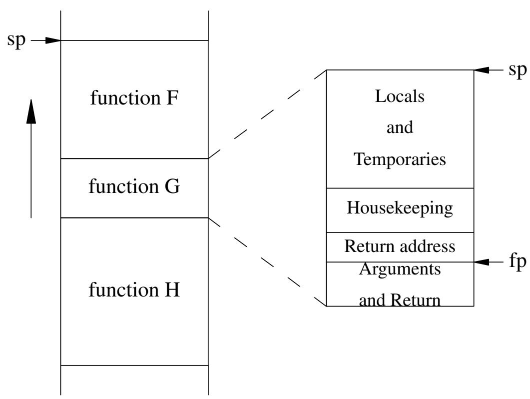


Figure 1000.1: Common storage organization of a function call stack.

Table 1000.1: Static count of number of calls: to functions defined within the same source file as the call, not defined in the file containing the call, and made via pointers-to functions. Parenthesized numbers are the corresponding dynamic count. Adapted from Chang, Mahlke, Chen, and Hwu.[36]

Name	Within File	Not in File	Via Pointer
cccp	191 (1,414)	4 (3)	1 (140)
compress	27 (4,283)	0 (0)	0 (0)
eqn	81 (6,959)	144 (33,534)	0 (0)
espresso	167 (55,696)	982 (925,710)	11 (60,965)
lex	110 (63,240)	234 (4,675)	0 (0)
tbl	91 (9,616)	364 (37,809)	0 (0)
xlist	331 (10,308,201)	834 (8,453,735)	4 (479,473)
yacc	118 (34,146)	81 (3,323)	0 (0)

Table 1000.2: Percentage of function invocations during execution of various programs in SPECint92. The column headed *Leaf* lists percentage of calls to leaf functions, *NonLeaf* calls to nonleaf functions (the issues surrounding this distinction are discussed elsewhere). The column headed *Direct* lists percentages of calls where a function name appeared in the expression, *Indirect* is where the function address was obtained via expression evaluation. Adapted from Calder, Grunwald, and Zorn.[27]

Program	Leaf	Non-Leaf	Indirect	Direct	Program	Leaf	NonLeaf	Indirect	Direct
burg	72.3	27.7	0.1	99.9	eqntott	85.3	14.7	68.7	31.3
ditroff	14.7	85.3	1.0	99.0	espresso	75.0	25.0	4.0	96.0
tex	20.0	80.0	0.0	100.0	gcc	28.9	71.1	5.4	94.6
xfig	35.5	64.5	6.2	93.8	li	13.4	86.6	2.9	97.1
xtex	50.6	49.4	3.0	97.0	sc	29.1	70.9	0.1	99.9
compress	0.1	99.9	0.0	100.0	Mean	38.6	61.4	8.3	91.7

Table 1000.3: Count of instructions executed and function calls made during execution of various SPECint92 programs on an Alpha AXP21064 processor. *Function calls invoked* includes indirect function calls; *Instructions/Call* is the number of instructions executed per call; *Total I-calls* is the number of indirect function calls made; and *Instructions/I-call* is the number of instructions executed per indirect call. Adapted from Calder, Grunwald, and Zorn.[27]

Program Name	Instructions Executed	Function Calls Invoked	Instructions/Call	Total I-calls	Instructions/I-call
burg	390,772,349	6,342,378	61.6	8,753	44,644.4
ditroff	38,893,571	663,454	58.6	6,920	5,620.5
tex	147,811,789	853,193	173.2	0	–
xfig	33,203,506	536,004	61.9	33,312	996.7
xtex	23,797,633	207,047	114.9	6,227	3,821.7
compress	92,629,716	251,423	368.4	0	–
eqntott	1,810,540,472	4,680,514	386.8	3,215,048	563.1
espresso	513,008,232	2,094,635	244.9	84,751	6,053.1
gcc	143,737,904	1,490,292	96.4	80,809	1,778.7
li	1,354,926,022	31,857,867	42.5	919,965	1,472.8
sc	917,754,841	12,903,351	71.1	13,785	66,576.3
dhrystone	608,057,060	18,000,726	33.8	0	–
Program mean	497,006,912	5,625,468	152.8	397,233	14,614.1

Table 1000.4: Mean and standard deviation of call stack depth during execution of various programs in SPECint92. Adapted from Calder, Grunwald, and Zorn.[27]

Program	Mean	Std. Dev.	Program	Mean	Std. Dev.
burg	10.5	30.84	eqntott	6.5	1.39
ditroff	7.1	2.45	espresso	11.5	4.67
tex	7.9	2.71	gcc	9.9	2.44
xfig	11.6	4.47	li	42.0	14.50
xtex	14.2	4.27	sc	6.8	1.41
compress	4.0	0.07	Mean	12.0	6.29

1001 The postfix expression denotes the called function.

Table 1001.1: Static count of functions defined, library functions called, direct and indirect calls to them and number of functions that had their addresses taken in SPECint95. Adapted from Cheng.[37]

Benchmark	Lines Code	Functions Defined	Library Functions	Direct Calls	Indirect Calls	& Function
008.espresso	14,838	361	24	2,674	15	12
023.eqntott	12,053	62	21	358	11	5
072.sc	8,639	179	53	1,459	2	20
085.cc1	90,857	1,452	44	8,332	67	588
124.m88ksim	19,092	252	36	1,496	3	57
126.gcc	205,583	2,019	45	19,731	132	229
130.li	7,597	357	27	1,267	4	190
132.ijpeg	29,290	477	18	1,016	641	188
134.perl	26,874	276	72	4,367	3	3
147.vortex	67,205	923	33	8,521	15	44

1002 The list of expressions specifies the arguments to the function.

Usage

Usage information on the number of arguments in calls to functions is given elsewhere.

289 function call number of arguments

1003 An argument may be an expression of any object type.

Usage

Information on parameter types is given elsewhere (see Table 1831.1).

Table 1003.1: Occurrence of various argument types in calls to functions (as a percentage of argument types in all calls). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
struct *	26.8	**void** *	4.0
int	16.5	**union** *	3.4
const char *	9.7	**unsigned char**	2.5
char *	8.4	**enum**	2.1
other-types	8.0	**unsigned short**	1.9
unsigned int	7.1	**const void** *	1.8
unsigned long	6.3	**long**	1.4

1004 In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.79)

function call preparing for

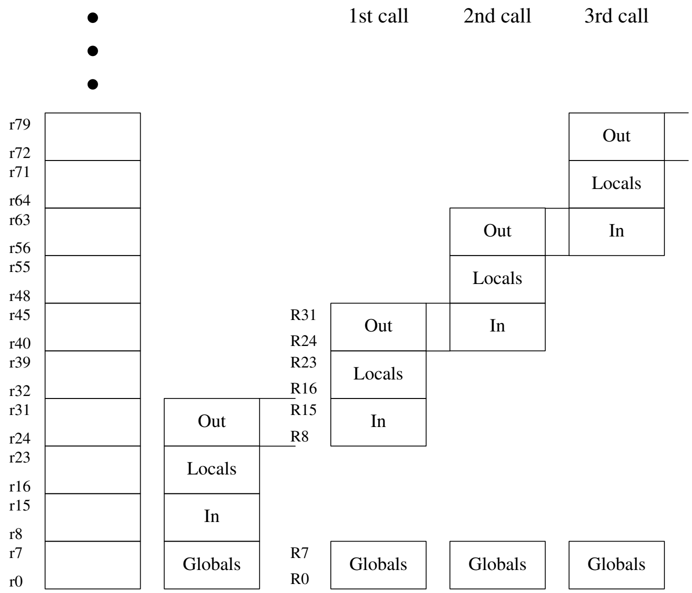


Figure 1004.1: A processor's register file (on the left) and a mapping to register windows for registers accessible to a program, after 0, 1, 2, and 3 *call* instructions have been executed. The mapping of the first eight registers is not affected by the *call* instruction.

If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. 1005

Table 1005.1: Occurrence of various return types in calls to functions (as a percentage of the return types of all function calls). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`void`	35.8	`union *`	3.2
`int`	30.5	`unsigned long`	3.1
`struct *`	9.1	`char *`	3.1
`void *`	6.3	`unsigned int`	2.1
other-types	5.2	`char`	1.6

---

function definition ends with ellipsis

If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. 1011

---

footnote 78

78) Most often, this is the result of converting an identifier that is a function designator. 1013

Usage

In most programs an identifier is converted in more than 99% of cases, although a lower percentage is occasionally seen (see Table 1001.1).

On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. 1015

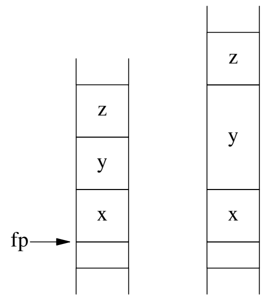


Figure 1011.1: An example of the impact, on relative stack addresses, of passing an argument having a type that occupies more storage than the declared parameter type. For instance, the offset of z, relative to the frame pointer *fp*, might be changed by passing an argument having a type different from the declared type of the parameter y (this can occur when there is no visible prototype at the point of call to cause the type of the argument to be converted).

Usage

Pointer types are the most commonly occurring kind of parameter type (see Table 1831.1).

1031 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. member selection

Table 1031.1: Number of member selection operators of the same object (number of dot selections is indicated down the left, and the number of indirect selections across the top). For instance, `x.m1->m2` is counted as one occurrence of the dot selection operator with one instance of the indirect selection operator. Based on the translated form of this book's benchmark programs.

. \ ->	0	1	2	3	4	5
0	0	165,745	10,396	522	36	4
1	28,160	34,065	3,437	230	7	0
2	3,252	6,643	579	26	0	0
3	363	309	35	5	0	0
4	16	33	2	0	0	0
5	0	15	0	0	0	0

1037 One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. union special guarantee

Usage

Measurements by Stiff et al.[159] of 1.36 MLOC (the SPEC95 benchmarks, gcc, binutils, production code from a Lucent Technologies' product and a few other programs) showed a total of 23,947 casts involving 2,020 unique types. For the `void *` ⇔ `struct *` conversion, they found 2,753 upcasts (610 unique types), 2,788 downcasts (606 unique types), and 538 cases (60 unique types) where there was no matching between the associated up/down casts. For the `struct *` ⇔ `struct *` conversions, they found 688 upcasts (78 unique types), 514 downcasts (66 unique types), and 515 cases (67 unique types) where there was no relationship associated with the types.

1053 **Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2).

Usage

No usage information is provided on compound literals because very little existing source code contains any use of them.

EXAMPLE string literals shared

EXAMPLE 6 Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example, 1076

```
(const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

Usage

In the visible source of the `.c` files 0.1% of string literals appeared as the operand of the equality operator (representing 0.3% of the occurrences of this operator).

unary-expression syntax

1080

```
unary-expression:
        postfix-expression
        ++ unary-expression
        -- unary-expression
        unary-operator cast-expression
        sizeof unary-expression
        sizeof ( type-name )
unary-operator: one of
        & * + - ~ !
```

Usage

postfix-expression syntax 985

See the Usage section of *postfix-expression* for `++` and `--` digraph percentages.

Table 1080.1: Common token pairs involving **sizeof**, *unary-operator*, prefix **++**, or prefix **--** (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
! defined	2.0	16.7	! (	14.5	0.5
*v --v	0.3	7.8	-v identifier	30.2	0.4
-v *floating-constant*	0.3	6.7	*v (	9.0	0.4
*v ++v	0.5	6.3	~ *integer-constant*	20.1	0.2
! --v	0.2	4.8	++v identifier	97.3	0.1
-v *integer-constant*	69.0	4.1	~ identifier	56.3	0.1
&v identifier	96.1	1.9	~ (	23.4	0.1
sizeof (	97.5	1.8	+v *integer-constant*	49.0	0.0
*v identifier	86.8	1.0	--v identifier	97.1	0.0
! identifier	81.9	0.8			

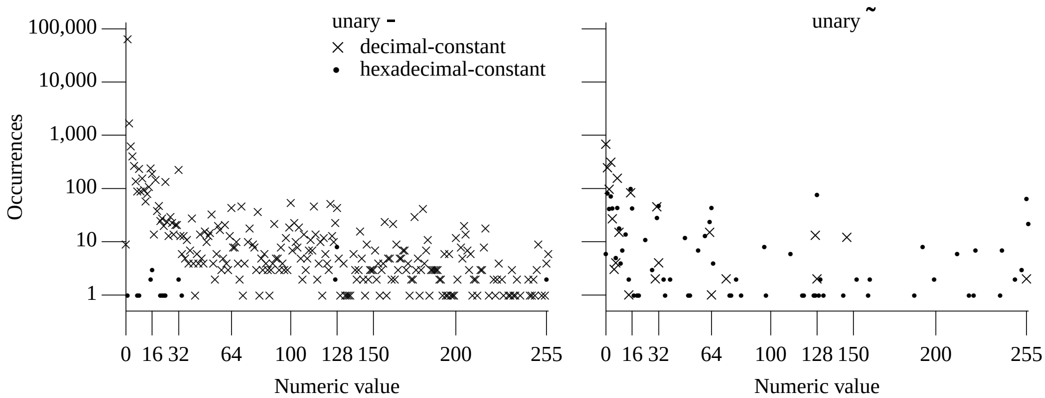


Figure 1080.1: Number of *integer-constant*s having a given value appearing as the operand of the unary minus and unary ~ operators. Based on the visible form of the `.c` files.

Table 1080.2: Occurrence of the *unary-operator*s, prefix `++`, and prefix `--` having particular operand types (as a percentage of all occurrences of the particular operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Operator	Type	%	Operator	Type	%	Operator	Type	%
`-v`	`_int`	96.0	`~`	`unsigned long`	6.8	`!`	`_long`	2.7
`*v`	ptr-to	95.3	`&v`	`int`	6.2	`~`	`unsigned char`	2.5
`+v`	`_int`	72.2	`~`	`unsigned int`	6.0	`&v`	`unsigned char`	2.4
`--v`	`int`	54.7	`+v`	`unsigned long`	5.6	`!`	`unsigned long`	2.1
`!`	`int`	50.0	`+v`	`long`	5.6	`~`	`long`	2.0
`~`	`_int`	49.3	`+v`	`float`	5.6	`++v`	`unsigned char`	1.9
`&v`	other-types	45.1	`!`	other-types	5.6	`~`	`_unsigned long`	1.7
`++v`	`int`	43.8	`++v`	`unsigned long`	5.2	`~`	`_unsigned int`	1.7
`++v`	ptr-to	33.3	`&v`	`struct *`	4.9	`!`	`unsigned char`	1.6
`~`	`int`	28.5	`--v`	`unsigned long`	4.7	`~`	other-types	1.6
`--v`	`unsigned int`	22.1	`!`	`unsigned int`	4.7	`-v`	`_double`	1.4
`!`	ptr-to	20.1	`*v`	fnptr-to	4.1	`-v`	other-types	1.3
`--v`	ptr-to	14.6	`&v`	`unsigned long`	4.0	`++v`	`long`	1.2
`&v`	`struct`	13.9	`--v`	other-types	4.0	`-v`	`int`	1.2
`&v`	`char`	13.1	`&v`	`long`	3.4	`!`	`_int`	1.2
`++v`	`unsigned int`	12.6	`&v`	`unsigned int`	3.0	`++v`	`unsigned short`	1.1
`+v`	`int`	11.1	`&v`	`unsigned short`	2.9	`&v`	`char *`	1.1
`!`	`char`	9.2	`!`	`enum`	2.9			

1095 The unary `*` operator denotes indirection. unary * indirection

Usage

A study by Mock, Das, Chambers, and Eggers[124] looked at how many different objects the same pointer dereference referred to during program execution (10 programs from the SPEC95 and SPEC2000 benchmarks were used). They found that in 90% to 100% of cases (average 98%) the set of objects pointed at, by a particular pointer dereference, contained one item. They also performed a static analysis of the source using a variety of algorithms for deducing points-to sets. On average (geometric mean) the static points to sets were 3.3 larger than the dynamic points to sets. SPEC benchmarks

1103 of the `!` operator, scalar type. ! operand type

Table 1103.1: Occurrence of the unary `!` operator in various contexts (as a percentage of all occurrences of this operator and the percentage of all occurrences of the given context that contains this operator). Based on the visible form of the `.c` files.

Context	% of `!`	% of Contexts
if control-expression	91.0	17.4
while control-expression	2.3	8.2
for control-expression	0.3	0.7
switch control-expression	0.0	0.0
other contexts	6.4	—

! equivalent to

The expression `!E` is equivalent to `(0==E)`. 1113

Usage

The visible form of the `.c` files contain 95,024 instances of the operator `!` (see Table 912.2 for information on punctuation frequencies) and 27,008 instances of the token sequence `== 0` (plus 309 instances of the form `== 0x0`). Integer constants appearing as the operand of a binary operator occur 28 times more often as the right operand than as the left operand.

sizeof constraints

The **sizeof** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. 1118

Table 1118.1: Occurrence of the **sizeof** operator having particular operand types (as a percentage of all occurrences of this operator). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
struct	48.2	**unsigned short**	2.7
[ ]	12.2	**struct** *	2.6
int	11.6	**char**	2.0
other-types	4.7	**unsigned char**	1.5
long	3.8	**char** *	1.5
unsigned int	3.6	**signed int**	1.2
unsigned long	3.4	**union**	1.1

cast-expression syntax

1133

```
cast-expression:
        unary-expression
        ( type-name ) cast-expression
```

Usage

Measurements by Stiff, Chandra, Ball, Kunchithapadam, and Reps[159] of 1.36 MLOC (SPEC95 version of gcc, binutils, production code from a Lucent Technologies product and a few other programs) showed a total of 23,947 casts involving 2,020 unique types. Of these 15,704 involved scalar types (not involving a structure, union, or function pointer) and 447 function pointer types. Of the remaining casts 7,796 (1,276 unique types) involved conversions between pointers to **void/char** and pointers to structure (in either direction) and 1,053 (209 unique types) conversions between pointers to structs.

cast scalar or void type

Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type. 1134

Usage

Usage information on implicit conversions is given elsewhere (see Table 653.1).

Table 1134.1: Occurrence of the cast operator having particular operand types (as a percentage of all occurrences of this operator). Based on the translated form of this book's benchmark programs.

To Type	From Type	%	To Type	From Type	%
(other-types)	other-types	40.1	(char *)	const char *	1.6
(void *)	_ int	18.9	(union *)	void *	1.5
(struct *)	struct *	11.2	(void)	long	1.3
(struct *)	_ int	4.2	(unsigned long)	unsigned long	1.3
(char *)	char *	4.0	(int)	int	1.3
(char *)	struct *	3.9	(unsigned int)	int	1.2
(struct *)	void *	2.8	(enum)	int:8 24	1.2
(unsigned char)	int	1.7	(char)	_ int	1.2
(struct *)	char *	1.7	(unsigned long)	ptr-to *	1.0

1143 multiplicative-expression syntax

```
multiplicative-expression:
        cast-expression
        multiplicative-expression * cast-expression
        multiplicative-expression / cast-expression
        multiplicative-expression % cast-expression
```

Table 1143.1: Percentage breakdown of errors in answers to multiplication problems. Figures are mean values for 60 adults tested on 2×2 to 9×9 from Campbell and Graham,[29] and 42 adults tested on 0×0 to 9×9 from Harley.[77] For the Campbell and Graham data, the operand error and operation error percentages are an approximation due to incomplete data.

	Campbell and Graham	Harley
Operand errors	79.1	86.2
Close operand errors	76.8	76.74
Frequent product errors	24.2	23.26
Table errors	13.5	13.8
Operation error	1.7	13.72
Error frequency	7.65	6.3

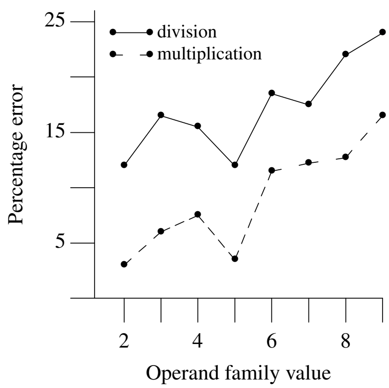


Figure 1143.1: Mean percentage of errors in simple multiplication (e.g., 3×7) and division (e.g., 81/9) problems as a function of the operand value (see paper for a discussion of the effect of the relative position of the minimum/maximum operand values). Adapted from Campbell.[28]

Table 1143.2: Common token pairs involving multiplicative operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: a consequence of the method used to perform the counts is that occurrences of the sequence *identifier* `*` are over estimated (e.g., occurrences of a typedef name followed by a `*` are included in the counts).

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier `*`	3.4	92.1	`/` **`sizeof`**	9.0	3.6
identifier `%`	0.0	57.7	`*` identifier	76.5	2.8
identifier `/`	0.1	54.3	`* )`	14.4	2.0
`) /`	0.3	33.9	*floating-constant* `/`	5.8	1.8
`) %`	0.1	31.8	`/` *integer-constant*	53.5	0.5
`*` *floating-constant*	0.2	12.5	`%` *integer-constant*	44.8	0.1
`*` **`sizeof`**	1.6	11.2	`/` identifier	27.5	0.1
integer-constant `/`	0.1	8.5	*floating-constant* `*`	6.8	0.1
`, %`	0.0	6.5	`/ (`	7.9	0.1
`/` *floating-constant*	2.0	6.4	`%` identifier	47.6	0.0
`* *v`	1.4	4.4			

1144

multiplicative operand type

Each of the operands shall have arithmetic type.

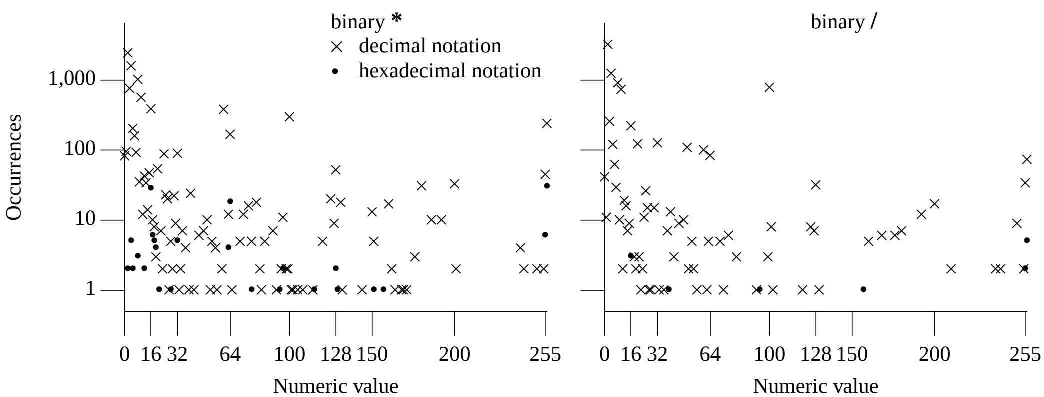


Figure 1143.2: Number of *integer-constant*s having a given value appearing as the right operand of the multiplicative operators. Based on the visible form of the `.c` files.

Table 1144.1: Occurrence of multiplicative operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	%	_int	40.6	_unsigned long	*	_int	2.8
int	/	_int	25.6	int	/	float	2.7
other-types	*	other-types	18.1	long	/	_int	2.5
other-types	/	other-types	16.2	unsigned long	%	int	2.3
_int	*	_int	13.4	_int	*	unsigned short	2.2
unsigned int	%	_int	12.6	_int	*	_unsigned long	2.2
int	%	int	12.2	_unsigned long	*	int	2.1
int	*	_int	12.1	unsigned long	*	_unsigned long	1.9
_int	/	_int	11.0	int	%	unsigned int	1.8
_unsigned long	/	_unsigned long	9.9	float	/	float	1.8
_unsigned long	*	unsigned char	9.5	_unsigned long	/	_int	1.6
int	*	_unsigned long	8.8	unsigned int	%	int	1.6
float	*	float	8.8	unsigned long	%	unsigned long	1.5
other-types	%	other-types	7.3	unsigned short	/	_int	1.3
unsigned long	/	_int	6.6	unsigned long	/	unsigned long	1.3
_int	*	int	6.5	unsigned int	*	_int	1.3
int	*	int	5.9	unsigned int	*	_unsigned long	1.2
unsigned long	/	_unsigned long	5.8	int	/	_unsigned long	1.2
unsigned int	/	_int	5.3	_double	/	_double	1.2
int	/	int	5.0	float	*	_int	1.1
unsigned int	%	unsigned int	4.2	unsigned long	*	_int	1.0
int	%	unsigned long	4.2	unsigned int	%	unsigned long	1.0
int	%	_unsigned long	3.9	int	/	unsigned long	1.0
long	%	_int	3.7	_int	*	unsigned int	1.0
unsigned long	%	_int	3.1				

1147 The result of the binary * operator is the product of the operands. binary * result

Usage

Measurements by Citron, Feitelson, and Rudolph[39] found that in a high percentage of cases the operands of multiplication operations repeat themselves (59% for integer operands and 43% for floating-point). Measurements were based on maintaining previous results in a 32-entry, 4-way associative, cache.

1148 The result of the / operator is the quotient from the division of the first operand by the second; binary / result

Usage

Measurements by Oberman[131] found that in a high percentage of cases division operations on floating-point operands repeat themselves (i.e., the same numerator and denominator values). The measurements were done using the SPECfp92 and NAS (Fortran-based) benchmarks.[12] Simulations using an infinite division operand cache found a hit rate (i.e., cache lookup could return the result of the division) of 69.8%, while a cache containing 128 entries had a hit rate of 60.9%. A more detailed analysis by Citron, Feitelson, and Rudolph[39] found a great deal of variability over different applications, with multimedia applications having hit rates of 50% (using a 32 entry, 4-way associative cache). SPEC benchmarks

1153 additive-expression syntax additive operators

```
additive-expression:
        multiplicative-expression
        additive-expression + multiplicative-expression
        additive-expression - multiplicative-expression
```

Table 1153.1: Common token pairs involving additive operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier +	1.0	77.5	+ **sizeof**	1.5	3.8
identifier -	0.5	75.7	+ *integer-constant*	33.7	1.9
) -	0.3	14.7	- *integer-constant*	44.0	1.3
) +	0.6	12.9	+ identifier	55.4	0.7
+ *floating-constant*	0.4	7.7	+ (	8.3	0.4
integer-constant +	0.4	6.3	- identifier	46.1	0.3
integer-constant -	0.2	5.8	- (	6.2	0.1

addition operand types

For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integer type. 1154

Table 1154.1: Occurrence of additive operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	+	**_int**	37.5	**unsigned long**	+	**_int**	2.6
int	-	**_int**	19.5	**unsigned long**	-	**unsigned long**	2.4
other-types	+	other-types	16.2	**unsigned int**	-	**unsigned int**	2.2
other-types	-	other-types	16.0	**long**	-	**_int**	2.2
_int	+	**_int**	11.8	**_int**	-	**int**	2.1
int	-	**int**	10.8	ptr-to	-	**_int**	2.0
_int	-	**_int**	8.8	**long**	-	**long**	2.0
ptr-to	-	ptr-to	6.4	**unsigned int**	+	**_int**	1.7
ptr-to	+	**unsigned long**	6.2	**float**	+	**float**	1.7
ptr-to	+	**long**	5.8	**unsigned short**	-	**int**	1.5
float	-	**float**	5.0	**unsigned long**	+	**unsigned long**	1.4
unsigned long	-	**_int**	4.9	**int**	-	**unsigned short**	1.4
int	+	**int**	4.7	**_int**	+	**int**	1.4
unsigned int	-	**_int**	4.2	**unsigned short**	+	**_int**	1.2
ptr-to	+	**int**	3.7	**unsigned short**	-	**_int**	1.1
_unsigned long	-	**_int**	3.1	**unsigned char**	-	**_int**	1.1
ptr-to	-	**unsigned long**	3.1	**unsigned int**	+	**unsigned int**	1.0
ptr-to	+	**_int**	3.0				

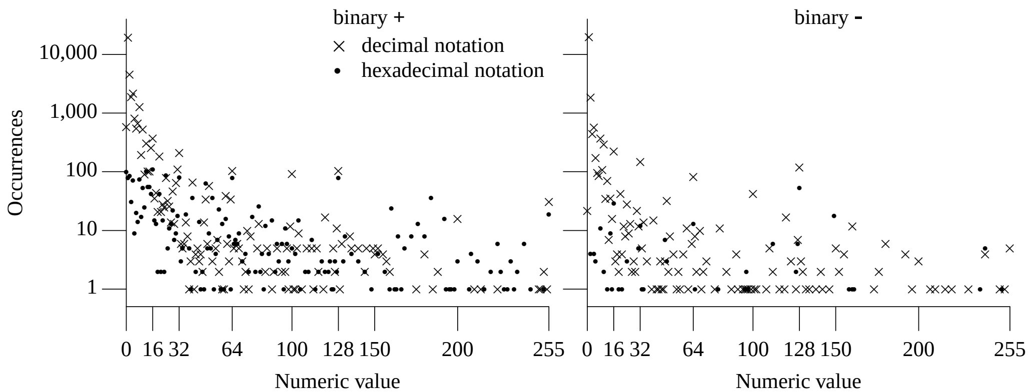


Figure 1153.1: Number of *integer-constant*s having a given value appearing as the right operand of additive operators. Based on the visible form of the `.c` files.

1159 — both operands are pointers to qualified or unqualified versions of compatible object types; or subtraction pointer operands

Table 1159.1: Occurrence of operands of the subtraction operator having a pointer type (as a percentage of all occurrences of this operator with operands having a pointer type). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
char *	-	**char ***	48.9	**void ***	-	**void ***	1.4
unsigned char *	-	**unsigned char ***	26.2	**int ***	-	**int ***	1.4
struct *	-	**struct ***	13.7	**unsigned short ***	-	**unsigned short ***	1.2
const char *	-	**const char ***	4.6	other-types	-	other-types	0.0

1160 — the left operand is a pointer to an object type and the right operand has integer type.

Table 1160.1: Occurrence of additive operators one of whose operands has a pointer type (as a percentage of all occurrences of each operator with one operand having a pointer type). Based on the translated form of this book's benchmark programs. Note: in the translator used the result of the **sizeof** operator had type **unsigned long**.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
char *	-	**unsigned long**	46.0	**unsigned char ***	-	**int**	1.7
char *	+	**unsigned long**	27.3	**const char ***	-	_ **int**	1.7
char *	+	**long**	26.8	**short ***	-	_ **int**	1.6
other-types	+	other-types	10.6	**char ***	+	**unsigned char**	1.6
char *	-	_ **int**	9.5	**char ***	-	**int**	1.6
struct *	-	array-index	9.1	**char ***	-	array-index	1.4
unsigned char *	-	_ **int**	8.8	**unsigned char ***	+	**unsigned int**	1.3
unsigned char *	+	_ **int**	7.4	**unsigned char ***	-	array-index	1.3
char *	+	**int**	6.6	**void ***	-	_ **int**	1.2
unsigned char *	+	**int**	5.7	**char ***	+	**signed int**	1.2
struct *	-	_ **int**	4.7	**unsigned long ***	+	**int**	1.1
char *	+	_ **int**	3.6	**struct ***	+	_ **int**	1.1
unsigned char *	-	_ **unsigned long**	2.1	**unsigned char ***	+	**unsigned short**	1.0
char *	+	**unsigned int**	1.9	**char ***	+	**unsigned short**	1.0
struct *	+	**int**	1.8	other-types	-	other-types	0.0

1163 The result of the binary + operator is the sum of the operands.

Table 1163.1: Mean square error in the result of summing, using five different algorithms, N values having a uniform or exponential distribution; where μ is the mean of the N values and σ^2 is the mean square error that occurs when two numbers are added.

Distribution	Increasing Order	Random Order	Decreasing Order	Insertion	Adjacency
Uniform $(0, 2\mu)$	$0.2\mu^2N^3\sigma^2$	$0.33\mu^2N^3\sigma^2$	$0.53\mu^2N^3\sigma^2$	$2.6\mu^2N^2\sigma^2$	$2.7\mu^2N^2\sigma^2$
Exponential (μ)	$0.13\mu^2N^3\sigma^2$	$0.33\mu^2N^3\sigma^2$	$0.63\mu^2N^3\sigma^2$	$2.63\mu^2N^2\sigma^2$	$4.0\mu^2N^2\sigma^2$

Usage

A study by Sweeney[161] dynamically analyzed the floating-point operands of the addition operator. In 26% of cases the values of the two operands were within a factor of 2 of each other, in 13% of cases within a factor of 4, and in 84% of cases within a factor of 1,024.

1181 shift-expression syntax

shift-expression:

additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

Table 1181.1: Common token pairs involving the shift operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier >>	0.1	63.9	] <<	0.5	5.3
identifier <<	0.1	37.3	<< *integer-constant*	63.4	0.8
integer-constant <<	0.5	36.1	>> *integer-constant*	79.8	0.7
) >>	0.2	28.0	<< identifier	28.4	0.1
) <<	0.2	20.3	<< (	8.1	0.1
] >>	0.4	6.2	>> identifier	15.9	0.0

Each of the operands shall have integer type. 1182

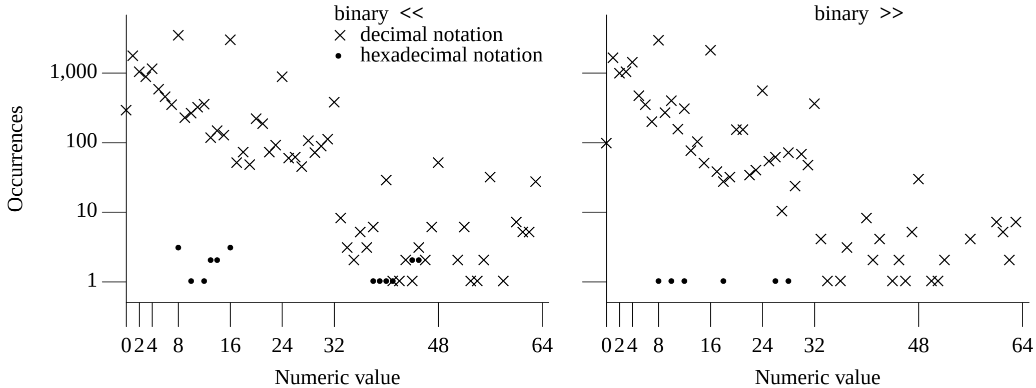


Figure 1181.1: Number of *integer-constant*s having a given value appearing as the right operand of the shift operators. Based on the visible form of the `.c` files.

Table 1182.1: Occurrence of shift operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	>>	**_int**	29.4	**unsigned char**	<<	**_int**	2.8
_int	<<	**_int**	27.1	**_long**	<<	**_long**	2.8
unsigned int	>>	**_int**	26.1	**unsigned int**	>>	**int**	2.6
_long	<<	**_int**	11.9	**_int**	>>	**_int**	2.5
int	<<	**_int**	11.8	**int**	>>	**int**	2.1
unsigned long	>>	**_int**	11.3	**long**	>>	**_int**	2.0
_int	<<	**int**	7.3	**unsigned long**	>>	**int**	1.8
unsigned short	>>	**_int**	7.0	**unsigned long**	<<	**_int**	1.8
other-types	>>	other-types	6.9	**long**	>>	**int**	1.7
int	<<	**int**	6.0	**_unsigned long**	<<	**int**	1.3
other-types	<<	other-types	5.8	**unsigned int**	>>	**unsigned int**	1.2
unsigned int	<<	**int**	5.3	**signed long**	>>	**_int**	1.2
_unsigned long	<<	**_int**	4.9	**unsigned short**	<<	**_int**	1.1
unsigned int	<<	**_int**	4.2	**long**	<<	**_int**	1.1
unsigned char	>>	**_int**	4.0	**int**	<<	**unsigned long**	1.1
unsigned long	<<	**int**	3.8				

1197 *relational-expression:* relational operators syntax

```
relational-expression:
        shift-expression
        relational-expression <  shift-expression
        relational-expression >  shift-expression
        relational-expression <= shift-expression
        relational-expression >= shift-expression
```

Table 1197.1: Common token pairs involving relational operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier <	0.7	87.9	>= *character-constant*	3.6	1.5
identifier >=	0.2	85.9	< *integer-constant*	40.0	1.3
identifier >	0.3	85.0	> *integer-constant*	53.2	0.9
identifier <=	0.1	84.8	>= *integer-constant*	41.2	0.4
) <=	0.1	10.4	< identifier	53.9	0.4
) >=	0.1	10.1	<= *integer-constant*	41.0	0.2
) <	0.3	9.9	> identifier	40.1	0.2
) >	0.1	9.6	>= identifier	50.0	0.1
<= *character-constant*	7.1	1.7	<= identifier	45.7	0.1

Table 1197.2: Occurrences (per million words) of English words used in natural language sentences expressing some relative state of affairs. Based on data from the British National Corpus.[108]

Word	Occurrences per Million Words	Word	Occurrences per Million Words
great	464	less	344
greater	154	lesser	18
greatest	51	least	45
greatly	33	–	–
–	–	less than	40

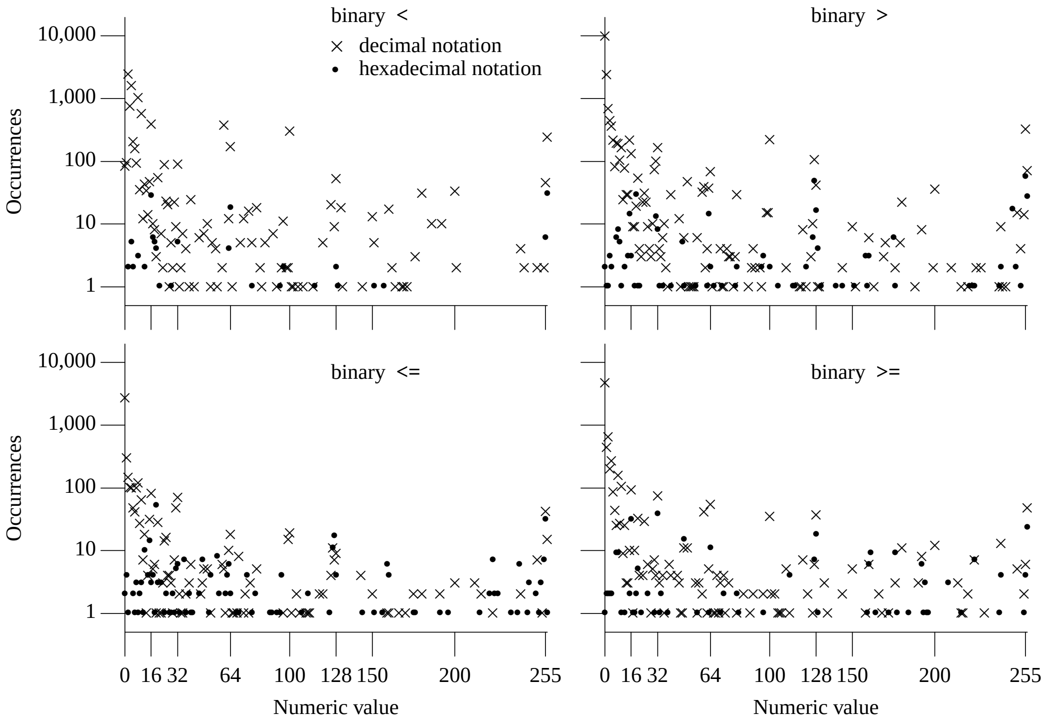


Figure 1197.1: Number of *integer-constant*s having a given value appearing as the right operand of relational operators. Based on the visible form of the `.c` files.

Table 1197.3: Occurrence of relational operators (as a percentage of all occurrences of the given operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	% of <	% of <=	% of >	% of >=
if control-expression	76.7 (3.4)	45.5 (6.7)	68.5 (1.8)	80.5 (6.0)
other contexts	11.5 (—)	4.8 (—)	9.5 (—)	8.4 (—)
while control-expression	4.8 (3.9)	4.6 (12.0)	4.8 (2.2)	7.6 (10.4)
for control-expression	7.1 (3.1)	45.2 (65.9)	17.2 (4.5)	3.5 (2.6)
switch control-expression	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

relational operators real operands

— both operands have real type;

Table 1199.1: Occurrence of relational operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	>=	**_int**	35.3	**unsigned char**	>	**_int**	2.3
int	>	**_int**	35.2	**unsigned char**	>=	**_int**	2.3
int	<	**_int**	34.8	ptr-to	<=	ptr-to	2.3
int	<=	**_int**	28.2	**unsigned int**	>=	**unsigned int**	2.1
int	<	**int**	25.5	**long**	<=	**long**	2.1
int	<=	**int**	17.5	**long**	>=	**_int**	2.0
other-types	>	other-types	15.8	**float**	>	**_int**	2.0
other-types	<	other-types	15.4	**unsigned long**	>	**unsigned long**	1.9
int	>	**int**	15.0	**unsigned short**	>	**unsigned short**	1.8
other-types	<=	other-types	14.5	**unsigned short**	>	**_int**	1.8
other-types	>=	other-types	13.2	**unsigned int**	<=	**unsigned int**	1.7
enum	<=	**_int**	12.6	ptr-to	>=	ptr-to	1.7
int	>=	**int**	10.8	**int**	<=	**unsigned long**	1.7
enum	>=	**enum**	7.5	**float**	>	**float**	1.7
unsigned int	>=	**int**	7.3	**char**	>=	**_int**	1.7
unsigned int	>	**_int**	6.0	**unsigned long**	>=	**unsigned long**	1.6
long	<	**_int**	5.3	**unsigned long**	>	**_int**	1.5
ptr-to	>	ptr-to	4.1	**double**	<=	**_double**	1.5
unsigned int	<=	**_int**	4.0	**unsigned long**	<=	**unsigned long**	1.4
unsigned int	<	**unsigned int**	3.7	**long**	>=	**long**	1.4
unsigned int	>=	**_int**	3.5	**int**	<	**unsigned long**	1.4
char	<=	**_int**	3.5	**unsigned long**	<	**unsigned long**	1.3
unsigned int	>	**unsigned int**	3.3	**long**	<	**long**	1.3
unsigned char	<=	**_int**	3.1	**_long**	>=	**_long**	1.3
long	>	**long**	2.9	**unsigned short**	<=	**unsigned short**	1.2
ptr-to	<	ptr-to	2.8	**unsigned int**	>	**int**	1.2
int	<	**unsigned int**	2.7	**float**	<	**_int**	1.2
unsigned long	<=	**_int**	2.6	**unsigned short**	<=	**_int**	1.1
unsigned int	<	**_int**	2.5	**unsigned char**	<	**_int**	1.1
_long	>=	**long**	2.5	**float**	<	**float**	1.1
long	>	**_int**	2.5	**unsigned long**	>	**int**	1.0
enum	>=	**_int**	2.5	**long**	>=	**int**	1.0
unsigned long	>=	**int**	2.4	**float**	<=	**_int**	1.0

1200

— both operands are pointers to qualified or unqualified versions of compatible object types; or

relational operators pointer operands

Table 1200.1: Occurrence of relational operators having particular operand pointer types (as a percentage of all occurrences of each operator with operands having a pointer type). Based on the translated form of this book's benchmark programs.

Left Operand	Op	Right Operand	%	Left Operand	Op	Right Operand	%
char *	>	char *	67.5	const char *	>	const char *	4.0
char *	<=	char *	39.6	other-types	>	other-types	3.8
char *	>=	char *	26.9	int *	>=	int *	3.6
char *	<	char *	25.8	const char *	>=	const char *	3.6
struct *	<=	struct *	23.2	struct *	>	struct *	3.1
unsigned char *	>=	unsigned char *	22.8	short *	<=	short *	3.0
unsigned char *	<	unsigned char *	21.0	other-types	<	other-types	2.8
short *	>=	short *	16.1	unsigned int *	>=	unsigned int *	2.6
struct *	<	struct *	14.9	const char *	<	const char *	2.6
unsigned char *	<=	unsigned char *	13.4	const unsigned char *	<	const unsigned char *	2.0
signed int *	<	signed int *	13.1	unsigned int *	>	unsigned int *	1.9
struct *	>=	struct *	13.0	unsigned long *	<=	unsigned long *	1.8
void *	>	void *	11.0	other-types	<=	other-types	1.8
void *	<	void *	9.4	const char *	<=	const char *	1.8
unsigned char *	>	unsigned char *	8.7	void *	>=	void *	1.6
unsigned short *	<=	unsigned short *	7.9	unsigned short *	<	unsigned short *	1.2
const unsigned char *	<=	const unsigned char *	4.9	unsigned int *	<	unsigned int *	1.2
ptr-to *	<	ptr-to *	4.8	union *	<=	union *	1.2
unsigned short *	>=	unsigned short *	4.7	int *	<	int *	1.2
const unsigned char *	>=	const unsigned char *	4.7	int *	<=	int *	1.2

equality operators syntax 1212

```
equality-expression:
        relational-expression
        equality-expression == relational-expression
        equality-expression != relational-expression
```

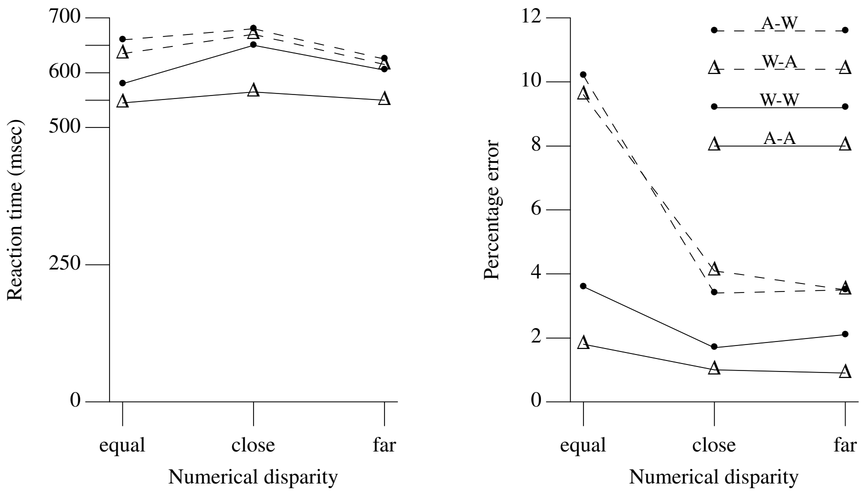


Figure 1212.1: Reaction time (in milliseconds) and error rates for same/different judgment for values between one and nine, expressed in *A*rabic or *W*ord form. Adapted from Dehaene and Akhavein.[53]

Table 1212.1: Common token pairs involving the equality operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have a very low percentage are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier !=	0.6	69.2	] !=	1.4	5.1
identifier ==	1.2	67.9	== -v	2.6	3.5
) ==	1.6	25.1	== *integer-constant*	25.5	2.0
) !=	0.8	24.7	== identifier	62.1	1.1
== *character-constant*	7.1	22.8	!= *integer-constant*	22.7	0.9
!= *character-constant*	5.3	8.4	!= identifier	65.0	0.6
] ==	3.1	5.6			

1214 — both operands have arithmetic type; equality operators arithmetic operands

Table 1214.1: Occurrence of equality operators having particular operand types (as a percentage of all occurrences of each operator). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
ptr-to	!=	ptr-to	28.5	**char**	!=	**_int**	3.9
int	==	**_int**	21.1	ptr-to	!=	**_int**	3.5
int	!=	**_int**	15.8	**unsigned long**	!=	**unsigned long**	2.5
ptr-to	==	ptr-to	15.3	**unsigned long**	!=	**_int**	2.2
other-types	==	other-types	12.7	**unsigned short**	!=	**_int**	2.0
other-types	!=	other-types	12.6	int:16 16	!=	**_int**	2.0
unsigned char	==	**_int**	9.5	**unsigned short**	!=	**unsigned short**	1.9
enum	==	**_int**	9.1	**unsigned int**	!=	**unsigned int**	1.9
int:16 16	==	**_int**	8.2	ptr-to	==	**_int**	1.8
int	!=	**int**	6.5	**unsigned short**	==	**_int**	1.7
int	==	**int**	6.5	**unsigned long**	==	**unsigned long**	1.7
char	==	**_int**	5.5	**unsigned long**	==	**_int**	1.6
unsigned char	!=	**_int**	4.8	**unsigned long**	!=	**_long**	1.3
enum	!=	**_int**	4.8	**unsigned char**	!=	**unsigned char**	1.3
unsigned int	!=	**_int**	4.4	**unsigned int**	==	**unsigned int**	1.1
unsigned int	==	**_int**	4.0				

1215 — both operands are pointers to qualified or unqualified versions of compatible types; equality operators pointer to compatible types

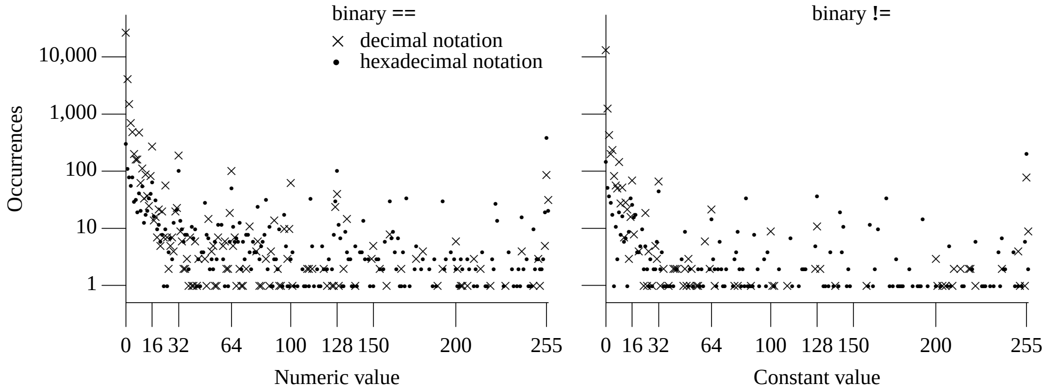


Figure 1212.2: Number of *integer-constants* having a given value appearing as the right operand of equality operators. Based on the visible form of the `.c` files.

Table 1215.1: Occurrence of equality operators having particular operand pointer types (as a percentage of all occurrences of each operator with operands having a pointer type; an _ prefix indicates a literal operand, *_int* is probably the 0 representation of the null-pointer constant). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
struct *	==	_ **int**	59.9	**int** *	!=	_ **int**	3.0
struct *	!=	_ **int**	52.2	**void** *	==	_ **int**	2.2
union *	!=	_ **int**	18.3	**const char** *	==	_ **int**	1.8
union *	==	_ **int**	18.1	**int**	==	**void** *	1.4
other-types	==	other-types	8.1	**const char** *	!=	_ **int**	1.4
char *	!=	_ **int**	8.1	**int**	!=	**void** *	1.3
char *	==	_ **int**	7.3	**unsigned char** *	==	_ **int**	1.1
array-index	!=	**void** *	6.9	ptr-to *	!=	_ **int**	1.1
other-types	!=	other-types	6.4	**char** *	!=	array-index	1.1

AND-expression syntax bitwise &

AND-expression: 1234

equality-expression

AND-expression & *equality-expression*

Table 1234.1: Occurrence of the **&** and **&&** operator (as a percentage of all occurrences of each operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the .c files.

Context	Binary &	&&
if control-expression	51.4 (10.5)	82.4 (10.4)
other contexts	45.3 (—)	7.7 (—)
while control-expression	2.1 (8.1)	6.9 (18.4)
for control-expression	0.3 (0.6)	3.0 (4.7)
switch control-expression	0.8 (5.2)	0.0 (0.0)

Table 1234.2: Common token pairs involving one of the operators **&**, **|**, or **^** (as a percentage of all occurrences of each token). Based on the visible form of the .c files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

<table>
<tr><th>Token Sequence</th><th>% Occurrence of First Token</th><th>% Occurrence of Second Token</th><th>Token Sequence</th><th>% Occurrence of First Token</th><th>% Occurrence of Second Token</th></tr>
<tr><td>identifier |</td><td>0.4</td><td>74.0</td><td>& identifier</td><td>57.1</td><td>0.6</td></tr>
<tr><td>identifier &</td><td>0.7</td><td>67.5</td><td>| identifier</td><td>79.8</td><td>0.4</td></tr>
<tr><td>identifier ^</td><td>0.0</td><td>51.1</td><td>& (</td><td>7.4</td><td>0.3</td></tr>
<tr><td>) ^</td><td>0.0</td><td>38.7</td><td>| (</td><td>14.4</td><td>0.3</td></tr>
<tr><td>& ~</td><td>4.6</td><td>30.1</td><td>^ *v</td><td>5.5</td><td>0.1</td></tr>
<tr><td>) &</td><td>1.1</td><td>27.7</td><td>| integer-constant</td><td>5.5</td><td>0.1</td></tr>
<tr><td>) |</td><td>0.4</td><td>20.8</td><td>^ integer-constant</td><td>20.8</td><td>0.0</td></tr>
<tr><td>] ^</td><td>0.0</td><td>5.1</td><td>^ identifier</td><td>55.5</td><td>0.0</td></tr>
<tr><td>] &</td><td>1.4</td><td>4.2</td><td>^ (</td><td>16.1</td><td>0.0</td></tr>
<tr><td>& integer-constant</td><td>30.6</td><td>1.5</td><td></td><td></td><td></td></tr>
</table>

& binary operand type

Each of the operands shall have integer type. 1235

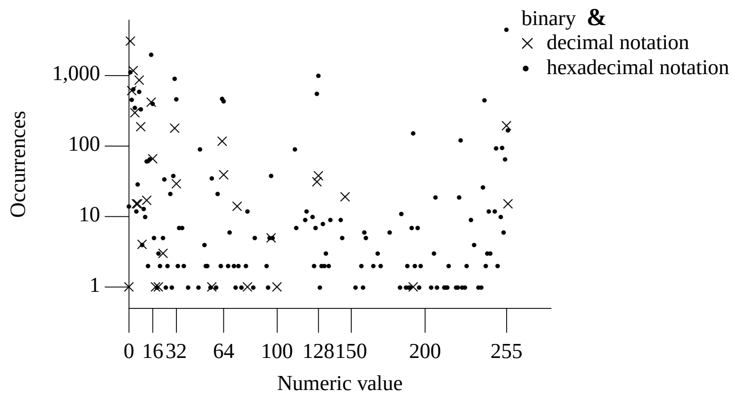


Figure 1234.1: Number of *integer-constant*s having a given value appearing as the right operand of the binary & operator. Based on the visible form of the .c files.

Table 1235.1: Occurrence of bitwise operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	|	_int	27.1	unsigned int	|	unsigned int	4.0
int	&	_int	24.3	unsigned long	&	_int	3.8
_int	|	_int	23.0	unsigned int	|	unsigned long	3.4
unsigned int	^	unsigned int	17.7	unsigned int	^	_int	3.3
other-types	&	other-types	13.9	unsigned int	^	int	3.1
int	|	int	13.6	unsigned long	&	int	2.6
_int	^	_int	13.5	long	^	long	2.6
unsigned long	^	unsigned long	12.2	unsigned char	&	int	2.5
unsigned int	&	_int	11.5	unsigned long	|	unsigned long	2.4
unsigned char	&	_int	10.3	unsigned long	&	unsigned long	2.0
int	^	_int	10.3	unsigned int	^	unsigned char	1.8
other-types	^	other-types	9.9	unsigned short	^	unsigned short	1.7
int	^	int	9.8	int	^	unsigned char	1.7
unsigned int	|	int	9.6	unsigned short	&	unsigned short	1.5
other-types	|	other-types	8.9	unsigned short	^	_int	1.5
unsigned short	&	_int	7.1	long	&	int	1.4
int	&	int	6.3	int	|	unsigned char	1.4
unsigned int	&	int	5.7	unsigned short	&	int	1.3
long	|	long	5.5	unsigned int	^	unsigned short	1.3
unsigned int	&	unsigned int	4.6	long	&	_int	1.2
unsigned char	^	unsigned char	4.6	_int	|	int	1.2
unsigned char	^	_int	4.2	int	^	unsigned short	1.1

1237 The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

	0	1
0	0	0
1	0	1

1240 *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression ^ *AND-expression*

Usage

The `^` operator represents 1.2% of all occurrences of bitwise operators in the visible source of the `.c` files.

The result of the `^` operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set). 1243

	0	1
0	0	1
1	1	0

inclusive-OR-expression syntax

1244

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*

Table 1244.1: Occurrence of the `|` and `||` operator (as a percentage of all occurrences of each operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	|	||
if control-expression	8.8 (0.7)	86.0 (6.9)
other contexts	90.7 (—)	11.9 (—)
while control-expression	0.3 (0.5)	1.9 (2.7)
for control-expression	0.0 (0.0)	0.3 (0.2)
switch control-expression	0.1 (0.3)	0.0 (0.0)

The result of the `|` operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set). 1247

	0	1
0	0	1
1	1	1

logical-AND-expression syntax

1248

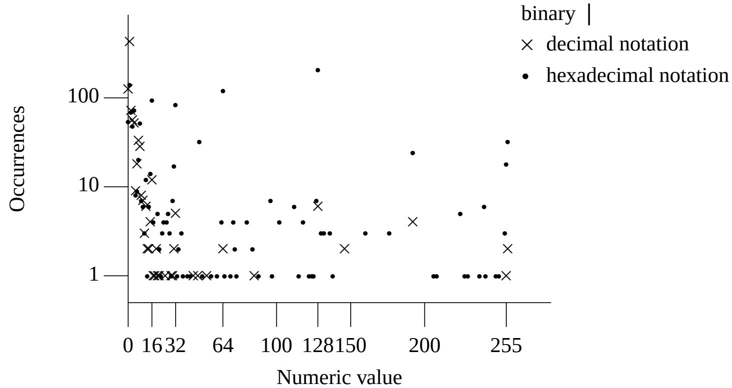


Figure 1244.1: Number of *integer-constant*s having a given value appearing as the right operand of the bitwise-OR operator. Based on the visible form of the `.c` files.

logical-AND-expression:
  inclusive-OR-expression
  logical-AND-expression **&&** *inclusive-OR-expression*

Table 1248.1: Various identities in boolean algebra expressed using the `||` and **&&** operators. Use of these identities may change the number of times a particular expression is evaluated (which is sometimes the rationale for rewriting it). The relative order in which expressions are evaluated may also change (e.g., when `A==1` and `B==0` in `(A && B) || (A && C)` the order of evaluation is A⇒ B⇒ A⇒ C, but after use of the distributive law the order becomes A⇒ B⇒ C).

<table>
<tr><th>Relative Order Preserved</th><th>Expression ⇒ Alternative Representation</th></tr>
<tr><td></td><td>Distributive laws</td></tr>
<tr><td>no</td><td><code>(A && B) || (A && C)</code> ⇒ <code>A && (B || C)</code></td></tr>
<tr><td>no</td><td><code>(A || B) && (A || C)</code> ⇒ <code>A || (B && C)</code></td></tr>
<tr><td></td><td>DeMorgan’s theorem</td></tr>
<tr><td>yes</td><td><code>!(A || B)</code> ⇒ <code>(!A) && (!B)</code></td></tr>
<tr><td>yes</td><td><code>!(A && B)</code> ⇒ <code>(!A) || (!B)</code></td></tr>
<tr><td></td><td>Other identities</td></tr>
<tr><td>yes</td><td><code>A && ((!A) || B)</code> ⇒ <code>A && B</code></td></tr>
<tr><td>yes</td><td><code>A || ((!A) && B)</code> ⇒ <code>A || B</code></td></tr>
<tr><td></td><td>The consensus identities</td></tr>
<tr><td>no</td><td><code>(A && B) || ((!A) && C) || (B && C)</code> ⇒ <code>(A && B) || ((!A) && C))</code></td></tr>
<tr><td>yes</td><td><code>(A && B) || (A && (!B) && C)</code> ⇒ <code>(A && B) || (A && C)</code></td></tr>
<tr><td>yes</td><td><code>(A && B) || ((!A) && C)</code> ⇒ <code>((!A) || B) && (A || C)</code></td></tr>
</table>

Table 1248.2: Common token pairs involving **&&**, or `||` (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

<table>
<tr><th>Token Sequence</th><th>% Occurrence of First Token</th><th>% Occurrence of Second Token</th><th>Token Sequence</th><th>% Occurrence of First Token</th><th>% Occurrence of Second Token</th></tr>
<tr><td>identifier &&</td><td>0.4</td><td>48.5</td><td>&& defined</td><td>0.9</td><td>6.2</td></tr>
<tr><td>) ||</td><td>0.9</td><td>42.7</td><td>|| !</td><td>11.3</td><td>6.0</td></tr>
<tr><td>identifier ||</td><td>0.2</td><td>39.3</td><td>character-constant ||</td><td>4.2</td><td>4.2</td></tr>
<tr><td>) &&</td><td>1.1</td><td>34.9</td><td>character-constant &&</td><td>5.3</td><td>3.3</td></tr>
<tr><td>|| defined</td><td>4.8</td><td>21.0</td><td>&& (</td><td>28.7</td><td>0.9</td></tr>
<tr><td>integer-constant ||</td><td>0.3</td><td>12.4</td><td>|| (</td><td>29.7</td><td>0.6</td></tr>
<tr><td>integer-constant &&</td><td>0.4</td><td>11.5</td><td>&& identifier</td><td>53.9</td><td>0.5</td></tr>
<tr><td>&& !</td><td>13.5</td><td>11.3</td><td>|| identifier</td><td>51.8</td><td>0.3</td></tr>
</table>

1249 Each of the operands shall have scalar type. && operand type

Table 1249.1: Occurrence of logical operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book’s benchmark programs.

<table>
<tr><th>Left Operand</th><th>Operator</th><th>Right Operand</th><th>%</th><th>Left Operand</th><th>Operator</th><th>Right Operand</th><th>%</th></tr>
<tr><td>int</td><td>||</td><td>int</td><td>87.7</td><td>_long</td><td>||</td><td>_long</td><td>2.2</td></tr>
<tr><td>int</td><td>&&</td><td>int</td><td>73.9</td><td>int</td><td>&&</td><td>ptr-to</td><td>2.2</td></tr>
<tr><td>other-types</td><td>&&</td><td>other-types</td><td>12.8</td><td>int</td><td>&&</td><td>char</td><td>1.8</td></tr>
<tr><td>other-types</td><td>||</td><td>other-types</td><td>8.4</td><td>int</td><td>||</td><td>_long</td><td>1.7</td></tr>
<tr><td>ptr-to</td><td>&&</td><td>int</td><td>4.5</td><td>int</td><td>&&</td><td>_int</td><td>1.3</td></tr>
<tr><td>char</td><td>&&</td><td>int</td><td>2.3</td><td>ptr-to</td><td>&&</td><td>ptr-to</td><td>1.1</td></tr>
</table>

1255 If the first operand compares equal to 0, the second operand is not evaluated. && second operand

Table 1255.1: Truth table showing how each operand of `(A || (B && C))` can affect its result. Case 1 and 2 show that `A` affects the outcome; Case 2 and 3 shows that `B` affects the outcome; Case 3 and 4 shows that `C` affects the outcome.

Case	A	B	C	Result
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

logical-OR-expression syntax

1256

```
logical-OR-expression:
        logical-AND-expression
        logical-OR-expression || logical-AND-expression
```

Usage

logical-AND-expression syntax 1248

Usage information is given elsewhere.

conditional-expression syntax

1264

```
conditional-expression:
        logical-OR-expression
        logical-OR-expression ? expression : conditional-expression
```

Table 1264.1: Common token pairs involving **?** or **:** (to prevent confusion with the **:** punctuation token the operator form is denoted by **?:**) (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
) ?	0.4	44.7	**?** *string-literal*	20.1	1.5
identifier **?**	0.1	44.0	**?:** *integer-constant*	28.7	0.3
identifier **?:**	0.1	40.3	**?** *integer-constant*	20.2	0.2
integer-constant **?:**	0.3	23.1	**?** identifier	43.9	0.1
string-literal **?:**	1.5	20.2	**?:** identifier	35.9	0.1
) ?:	0.1	11.6	**?: (**	7.2	0.1
integer-constant **?**	0.1	9.6	**? (**	6.2	0.1
?: *string-literal*	21.0	1.6			

conditional operator second and third operands

One of the following shall hold for the second and third operands: 1266

Table 1266.1: Occurrence of the ternary **:** operator (denoted by the character sequence **?:**) having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
ptr-to	**?:**	ptr-to	29.5	**int**	**?:**	**_int**	5.7
other-types	**?:**	other-types	12.1	**_char**	**?:**	**_char**	3.4
_int	**?:**	**_int**	10.4	**unsigned int**	**?:**	**unsigned int**	2.2
int	**?:**	**int**	10.0	**unsigned short**	**?:**	**unsigned short**	1.2
void	**?:**	**void**	9.4	**signed int**	**?:**	**_int**	1.1
unsigned long	**?:**	**unsigned long**	7.9	**char**	**?:**	**void**	1.1
_int	**?:**	**int**	6.0				

1288 *assignment-expression:* assignment-expression syntax

conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of

= *= /= %= += -= <<= >>= &= ^= |=

Usage

For a comparison with load frequencies see Table 976.2.

Table 1288.1: Common token pairs involving the assignment operators (as a percentage of all occurrences of each token). Based on the visible form of the .c files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier %=	0.0	100.0	v++ =	7.6	0.7
identifier /=	0.0	99.3	+= *integer-constant*	21.7	0.3
identifier >>=	0.0	99.3	|= identifier	77.0	0.2
identifier <<=	0.0	97.5	+= identifier	68.0	0.2
identifier +=	0.3	96.3	>>= *integer-constant*	87.1	0.1
identifier *=	0.0	96.0	-= *integer-constant*	24.2	0.1
identifier -=	0.1	95.2	&= *integer-constant*	12.4	0.1
identifier |=	0.3	93.9	|= *integer-constant*	10.7	0.1
identifier &=	0.1	93.1	-= identifier	65.1	0.1
identifier =	9.4	90.9	+= (	6.5	0.1
identifier ^=	0.0	85.9	|= (	12.0	0.1
&= ~	75.0	52.5	<<= *integer-constant*	85.1	0.0
= +v	0.0	45.1	/= *integer-constant*	52.1	0.0
= *floating-constant*	0.1	15.7	*= *integer-constant*	39.8	0.0
= *character-constant*	0.8	14.2	^= *integer-constant*	34.5	0.0
= -v	1.6	12.0	%= *integer-constant*	31.5	0.0
] ^=	0.0	11.1	&= identifier	8.6	0.0
= &v	1.9	10.2	%= identifier	68.1	0.0
= *v	1.1	9.9	^= identifier	46.4	0.0
= *integer-constant*	19.6	9.0	*= identifier	44.2	0.0
] =	21.8	6.8	/= identifier	34.6	0.0
= identifier	62.5	6.5	<<= identifier	13.4	0.0
= **sizeof**	0.3	5.9	>>= identifier	10.5	0.0
] &=	0.2	5.7	**#error** =	16.9	0.0
] |=	0.4	4.6	-= (	7.0	0.0
= (	9.1	3.5	/= (	5.8	0.0
*= *floating-constant*	6.3	1.6	^= (	13.9	0.0

Table 1288.2: Occurrence of executed store instructions (as a percentage of all instructions executed) in two different kinds of functions (*Leaf* functions do not call any other functions, while *Non-Leaf* do). Adapted from Calder, Grunwald, and Zorn.[27]

Program	Leaf	Non-Leaf	Program	Leaf	Non-Leaf
burg	34.3	7.7	eqntott	0.0	11.4
ditroff	8.3	8.3	espresso	6.5	3.9
tex	15.1	9.8	gcc	9.6	12.0
xfig	8.0	11.7	li	0.0	16.3
xtex	8.3	11.2	sc	1.2	11.1
compress	83.5	9.2	Mean	15.9	10.2

An assignment operator stores a value in the object designated by the left operand. 1290

Usage

value locality

A study by Lepak, Bell, and Lipasti[110] investigated value locality with respect to store operations (using the SPEC95 benchmarks). They defined a *silent store* to be a store operation that does not change the system state (i.e., the value being written matches the value already held at the location being stored to). They also defined *program structure store value locality* (PSSVL) to refer to the same value being stored from the same program location and *message-passing store value locality* (MPSVL) to refer to the same value being stored to the same address in storage (which may be holding different objects at different times during the execution of a program).

Table 1290.1: Percentage of stores that are *silent*. The results from two instruction sets, the PowerPC (PPC) and SimpleScalar (SS), are given for silent stores. The measurements for Program Structure Store Value Locality (PSSVL) and Message-Passing Store Value Locality (MPSVL) are for the PowerPC only. Adapted from Lepak, Bell, and Lipasti.[110]

Program	Silent stores (PPC/SS)	PSSVL (PPC)	MPSVL (PPC)	Program	Silent stores (PPC/SS)	PSSVL (PPC)	MPSVL (PPC)
go	38/27	30	36	tomcatv	47/33	40	45
m88ksim	68/62	56	65	swim	34/26	20	19
gcc	53/46	37	49	mgrid	23/ 7	24	17
compress	42/39	35	16	applu	37/35	35	28
li	34/20	32	34	apsi	21/25	22	20
ijpeg	43/33	52	46	fpppp	15/15	15	14
perl	49/36	39	42	wave5	25/22	30	20
vortex	64/55	71	57				

compound assignment constraints

For the operators += and -= only, either the left operand shall be a pointer to an object type and the right shall have integer type, or the left operand shall have qualified or unqualified arithmetic type and the right shall have arithmetic type. 1310

Table 1310.1: Occurrence of assignment operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand, occurrences below 2.3% were counted as *other-types*). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
other-types	-=	other-types	34.5	float	/=	float	6.4
other-types	+=	other-types	33.5	unsigned short	|=	_int	6.2
other-types	=	other-types	32.8	ptr-to	+=	_int	6.2
int	%=	_int	31.0	unsigned long	|=	int	6.1
ptr-to	=	ptr-to	29.7	unsigned int	-=	unsigned int	5.9
int	*=	_int	29.5	unsigned short	>>=	_int	5.8
long	-=	long	28.9	unsigned char	<<=	_int	5.7
unsigned int	<<=	_int	28.3	other-types	%=	other-types	5.7
unsigned int	>>=	_int	28.2	long	+=	_int	5.6
unsigned int	^=	unsigned int	26.7	long	*=	_int	5.3
int	>>=	_int	26.2	unsigned long	&=	int	5.1
int	<<=	_int	25.5	unsigned long	/=	_int	5.0
int	/=	_int	23.8	unsigned int	&=	unsigned int	4.6
int	+=	int	22.1	unsigned int	|=	unsigned int	4.6
unsigned char	&=	int	19.7	long	%=	_int	4.6
unsigned int	&=	int	19.4	unsigned short	/=	_int	4.5
int	-=	int	17.4	unsigned char	&=	_int	4.3
long	^=	long	16.9	unsigned long	|=	_int	4.1
other-types	*=	other-types	16.8	unsigned char	|=	int	3.9
other-types	&=	other-types	16.7	long	<<=	_int	3.8
int	&=	int	16.2	float	*=	_double	3.7
unsigned long	<<=	_int	15.9	unsigned int	+=	unsigned int	3.5
other-types	^=	other-types	15.3	long	&=	int	3.5
other-types	/=	other-types	14.4	unsigned int	=	unsigned int	3.4
other-types	|=	other-types	13.5	int	%=	unsigned int	3.4
unsigned int	/=	_int	12.9	unsigned long	^=	int	3.3
ptr-to	+=	int	12.8	float	*=	double	3.3
unsigned int	%=	_int	12.6	unsigned long	*=	_int	3.1
int	%=	int	12.6	unsigned char	^=	unsigned char	3.1
int	=	int	12.3	unsigned char	^=	int	3.1
unsigned int	|=	_int	12.1	ptr-to	+=	unsigned long	3.1
float	*=	float	12.1	double	*=	double	3.1
int	|=	_int	12.0	unsigned short	/=	unsigned short	3.0
unsigned char	|=	_int	11.7	unsigned short	|=	int	3.0
unsigned int	%=	unsigned int	11.5	int	/=	unsigned int	3.0
unsigned char	%=	_int	11.5	float	/=	int	3.0
int	/=	int	11.4	double	/=	double	3.0
unsigned long	^=	unsigned long	11.3	unsigned int	+=	_int	2.9
int	^=	_int	11.1	float	*=	_int	2.9
int	=	_int	11.0	unsigned long	+=	unsigned long	2.8
unsigned char	>>=	_int	10.3	unsigned long	|=	unsigned long	2.8
other-types	>>=	other-types	9.6	unsigned long	|=	long	2.8
unsigned long	>>=	_int	9.5	long	=	long	2.8
int	*=	int	9.3	int	&=	_int	2.8
unsigned short	<<=	_int	8.9	float	=	float	2.8
unsigned int	*=	_int	8.4	unsigned int	-=	int	2.7
int	-=	_int	8.0	int	>>=	int	2.7
unsigned short	&=	int	7.9	int	^=	int	2.7
long	>>=	_int	7.7	unsigned char	=	_int	2.6
unsigned int	|=	int	7.5	float	-=	float	2.6
long	/=	_int	7.4	unsigned long	=	unsigned long	2.5
int	+=	_int	7.4	unsigned long	<<=	unsigned int	2.5
int	|=	int	7.4	int	<<=	int	2.5
unsigned short	%=	_int	6.9	float	/=	_double	2.5
other-types	<<=	other-types	6.7	int	*=	float	2.4
unsigned char	^=	_int	6.4	unsigned char	|=	unsigned char	2.3

declaration syntax

1348

```
declaration:
        declaration-specifiers init-declarator-list_opt ;
declaration-specifiers:
        storage-class-specifier declaration-specifiers_opt
        type-specifier declaration-specifiers_opt
        type-qualifier declaration-specifiers_opt
        function-specifier declaration-specifiers_opt
init-declarator-list:
        init-declarator
        init-declarator-list , init-declarator
init-declarator:
        declarator
        declarator = initializer
```

Table 1348.1: Occurrence of types used in declarations of objects (as a percentage of all types). Adapted from Engblom[60] and this book's benchmark programs.

Type	Embedded	Book's benchmarks
integer	55.97	37.5
float	0.05	1.6
pointer	22.08 (data)/0.23 (code)	48.2
struct/union	9.88	6.1
array	11.80	6.6

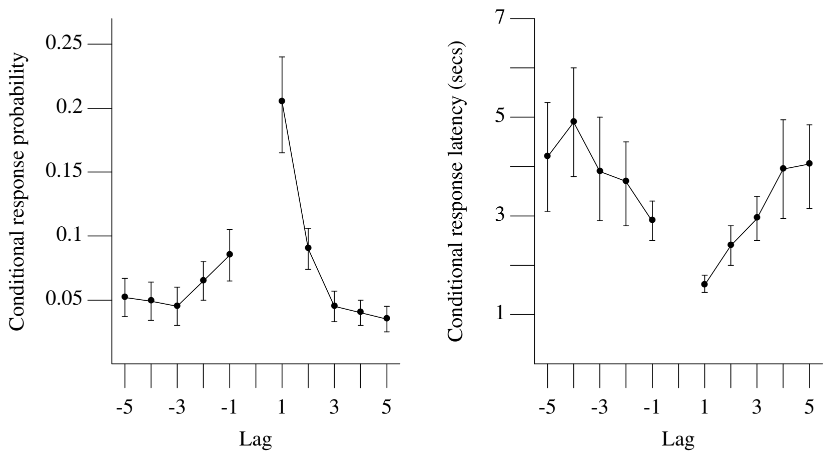


Figure 1348.1: The *lag recency effect*. The plot on the left shows the probability of a subject recalling an item having a given lag, while the plot on the right gives the time interval between recalling an item having a given lag (error bars give 95% confidence interval). If a subject, when asked to remember the list "ABSENCE HOLLOW PUPIL", recalled "HOLLOW" then "PUPIL", the recall of "PUPIL" would have a lag of one ("ABSENCE" followed by "PUPIL" would be a lag of 2). Had the subject recalled "HOLLOW" then "ABSENCE", the recall of "ABSENCE" would be a lag of minus one. Adapted from Howard and Kahana.[85]

Table 1348.2: Occurrence of types used to declare objects in block scope (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	28.1	**long**	3.0
struct *	27.7	**union** *	2.9
other-types	10.8	**unsigned short**	2.3
unsigned int	5.5	**unsigned char**	2.0
struct	4.9	**char**	1.8
unsigned long	4.8	**char** []	1.5
char *	3.5	**unsigned char** *	1.3

Table 1348.3: Occurrence of types used to declare objects with internal linkage (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	20.9	**const char** []	2.4
other-types	14.4	**unsigned int**	1.8
struct	13.0	**const struct**	1.8
struct *	8.2	**void** *()	1.7
struct []	7.4	**const unsigned char** []	1.6
(const char * const) []	4.0	**unsigned int** []	1.4
unsigned char []	3.4	**int** *()	1.4
unsigned short []	3.3	**(struct *)** []	1.3
int []	2.9	**(char *)** []	1.3
char *	2.8	**unsigned long**	1.2
char []	2.7	**const short** []	1.2

Table 1348.4: Occurrence of types used to declare objects with external linkage (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	22.8	**char** *	3.2
const char []	15.4	**union** *	3.0
other-types	10.6	**enum**	2.4
struct *	10.3	**float**	1.4
const struct	10.2	**char** []	1.4
struct	8.2	**unsigned int**	1.2
void *()	4.6	**int** []	1.2
struct []	4.1		

1364 storage-class specifier syntax

storage-class-specifier:
typedef
extern
static
auto
register

Table 1364.1: Common token pairs involving a *storage-class*. Based on the visible form of the `.c` files (the keyword **auto** occurred 14 times).

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
static void	33.7	32.7	**extern int**	32.1	1.7
static int	28.2	15.1	**register struct**	19.1	1.4
typedef union	3.2	11.0	**typedef struct**	62.4	1.2
static const	1.5	10.0	**register int**	23.0	1.2
static volatile	0.3	8.6	**register char**	10.2	1.2
typedef enum	10.8	8.2	**register unsigned**	6.1	0.9
static signed	0.0	6.5	**extern char**	7.4	0.9
static unsigned	3.8	5.5	**extern struct**	6.9	0.5
extern double	1.3	5.5	**static** identifier	21.0	0.3
static char	4.1	5.1	**typedef unsigned**	6.2	0.2
static struct	6.4	4.8	**typedef** identifier	7.9	0.0
register enum	1.6	4.6	**register** identifier	35.9	0.0
extern void	21.5	2.1	**extern** identifier	23.7	0.0

Table 1364.2: Common token pairs involving a *storage-class*. Based on the visible form of the `.h` files (the keyword **auto** occurred 6 times).

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
typedef union	12.4	67.1	**typedef unsigned**	6.6	3.1
typedef enum	6.2	37.2	**extern unsigned**	2.9	2.8
typedef signed	0.5	28.6	**static void**	10.3	2.2
extern void	28.6	24.0	**typedef void**	4.0	1.6
extern double	0.3	17.9	**static int**	7.0	1.2
typedef struct	46.3	16.6	**extern** identifier	32.2	0.9
extern int	23.2	15.2	**register long**	16.0	0.8
extern float	0.3	9.8	**register unsigned**	24.8	0.6
register signed	2.6	8.2	**static** identifier	70.3	0.5
static const	6.4	5.0	**register int**	18.4	0.3
extern char	3.8	4.8	**typedef** identifier	16.7	0.2
extern struct	4.3	3.3	**register** identifier	18.4	0.0

register storage-class

A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. 1369

Table 1369.1: Degree of use of floating-point and integer register instances (a particular value loaded into a register). Values denote the percentage of register instances with a particular degree of use (listed across the top), for the program listed on the left. For instance, 15.51% of the integer values loaded into a register, in `gcc`, are used twice. Left half of table refers to floating-point register instances, right half of table to integer register instances. Zero uses of a value loaded into a register occur in situations such as an argument passed to a function that is never accessed. Adapted from Franklin and Sohi.[70]

Usage	0	1	2	3	≥4	Average	0	1	2	3	≥4	Average
eqntott							0.89	71.34	17.54	9.47	0.76	1.86
espresso							3.67	72.30	17.66	3.74	2.63	1.48
gcc							6.26	67.37	15.51	4.45	6.41	1.69
xlisp							4.27	66.14	12.42	10.20	6.97	1.84
dnasa7	0.00	99.83	0.02	0.03	0.12	1.31	0.67	2.36	16.29	64.36	16.33	3.28
doduc	1.46	84.00	9.51	1.94	3.09	1.36	10.31	44.35	26.52	10.13	8.69	2.93
fpppp	0.16	91.09	6.15	1.14	1.46	1.16	1.34	10.12	83.45	0.46	4.63	3.09
matrix300	0.00	99.92	0.00	0.00	0.08	1.25	15.29	61.54	7.71	0.12	15.35	1.92
spice2g6	0.21	79.85	19.22	0.16	0.56	1.22	4.04	73.38	12.08	3.56	6.94	1.68
tomcatv	0.00	86.43	8.30	1.49	3.77	1.26	0.12	24.99	37.54	27.40	9.96	3.22

1378

type specifier syntax

type-specifier:

```
void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
```

~~**_Imaginary**~~

struct-or-union-specifier
enum-specifier
typedef-name

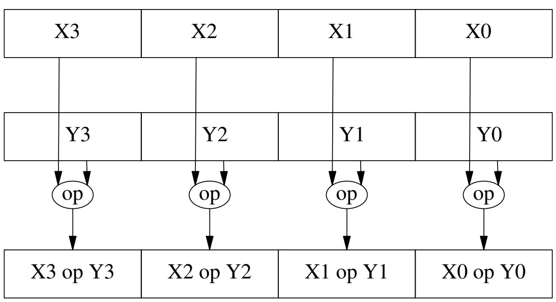


Figure 1378.1: Behavior of packed single-precision floating-point operations supported by the Intel Pentium processor.[87]

Table 1378.1: Common token pairs involving a *type-specifier*. Based on the visible form of the `.c` files. The type specifiers **_Bool**, **_Complex**, and **_Imaginary** did not appear in the visible form of the `.c` files.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token
unsigned long	38.7	72.2
unsigned short	5.8	63.8
char *p	74.5	63.3
(**signed**	0.0	60.5
; **enum**	0.1	45.5
(**struct**	2.9	41.8
; **float**	0.1	40.0
; **union**	0.0	33.7
static void	33.7	32.7
(**float**	0.0	32.0
(**unsigned**	1.0	29.0
(**void**	1.4	26.6
; **unsigned**	1.0	26.4
; **int**	2.5	24.8
(**char**	1.0	23.9
{ **union**	0.0	23.4
(**double**	0.0	22.9
; **double**	0.0	19.8
void *p	17.5	19.0
, **unsigned**	0.6	18.9
} **void**	4.1	18.0
unsigned char	21.2	18.0
; **struct**	1.3	17.6
; **char**	0.8	17.5
, **int**	1.4	15.9
static int	28.2	15.1
; **signed**	0.0	14.7
{ **struct**	4.3	14.5
identifier **double**	0.0	13.1
{ **unsigned**	1.9	12.5
, **struct**	0.8	12.2
{ **int**	4.8	11.5
{ **enum**	0.1	11.1
typedef union	3.2	11.0
(**short**	0.0	11.0
; **short**	0.0	10.6
(**int**	1.0	10.6
, **float**	0.0	10.6
const char	54.1	10.4
{ **float**	0.1	10.2
(**union**	0.0	9.9
, **char**	0.4	9.9
(**long**	0.2	9.2
, **enum**	0.0	9.2
unsigned int	24.6	9.1
{ **double**	0.0	8.6
typedef enum	10.8	8.2
, **double**	0.0	8.2
int *p	4.1	8.1
, **union**	0.0	8.0
, **signed**	0.0	7.9
) **enum**	0.0	7.1
{ **char**	1.3	7.1
static signed	0.0	6.5
; **void**	0.3	6.3
; **long**	0.1	6.2
, **void**	0.3	5.8
static unsigned	3.8	5.5
extern double	1.3	5.5
} **int**	2.2	5.3
{ **signed**	0.0	5.2
static char	4.1	5.1
header-name **double**	0.2	5.1
static struct	6.4	4.8
register enum	1.6	4.6
long *p	7.1	2.8
int identifier	87.6	2.3
extern void	21.5	2.1
struct identifier	99.0	1.9
extern int	32.1	1.7
short *p	21.8	1.4
register struct	19.1	1.4
const unsigned	6.2	1.4
const struct	11.1	1.3
typedef struct	62.4	1.2
register int	23.0	1.2
register char	10.2	1.2
volatile unsigned	25.6	1.1
void identifier	61.7	0.9
void)	17.5	0.9
register unsigned	6.1	0.9
extern char	7.4	0.9
const void	5.3	0.8
signed short	11.3	0.7
int)	6.6	0.6
extern struct	6.9	0.5
volatile struct	15.5	0.4
long identifier	68.3	0.4
long)	21.7	0.4
float *p	9.2	0.3
char identifier	22.6	0.3
typedef unsigned	6.2	0.2
signed long	20.8	0.2
double *p	7.9	0.2
volatile int	7.4	0.1
unsigned identifier	7.0	0.1
union {	34.5	0.1
signed char	22.6	0.1
short identifier	60.9	0.1
enum {	13.4	0.1
union identifier	65.5	0.0
signed int	7.5	0.0
signed)	37.9	0.0
short)	14.0	0.0
float identifier	64.3	0.0
float)	26.1	0.0
enum identifier	86.6	0.0
double identifier	70.7	0.0
double)	19.1	0.0

1382 type specifiers possible sets of

--~ **void**
--~ **char**
--~ **signed char**
--~ **unsigned char**
--~ **short**, **signed short**, **short int**, or **signed short int**
--~ **unsigned short**, or **unsigned short int**
--~ **int**, **signed**, or **signed int**
--~ **unsigned**, or **unsigned int**
--~ **long**, **signed long**, **long int**, or **signed long int**
--~ **unsigned long**, or **unsigned long int**
--~ **long long**, **signed long long**, **long long int**, or **signed long long int**
--~ **unsigned long long**, or **unsigned long long int**
--~ **float**
--~ **double**
--~ **long double**
--~ **_Bool**
--~ **float _Complex**
--~ **double _Complex**
--~ **long double _Complex**
~~--~**float _Imaginary**~~
~~--~**double _Imaginary**~~
~~--~**long double _Imaginary**~~
--~ struct or union specifier
--~ enum specifier
--~ typedef name

Table 1382.1: Occurrence of *type-specifier* sequences (as a percentage of all type specifier sequences; cut-off below 0.1%). Based on the visible form of the `.c` files.

Type Specifier Sequence	%	Type Specifier Sequence	%
int	39.9	**long**	2.2
void	24.3	**unsigned**	1.6
char	15.6	**unsigned short**	0.9
unsigned long	6.2	**float**	0.6
unsigned int	4.0	**short**	0.5
unsigned char	3.4	**double**	0.5

1390 struct/union syntax

struct-or-union-specifier:
    struct-or-union identifier$_{opt}$ **{** *struct-declaration-list* **}**
    struct-or-union identifier
struct-or-union:
    struct
    union
struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration
struct-declaration:
    specifier-qualifier-list struct-declarator-list **;**
specifier-qualifier-list:

type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list **,** *struct-declarator*
struct-declarator:
declarator
declarator$_{opt}$ **:** *constant-expression*

Usage

A study by Sweeney and Tip[162] of C++ applications found that on average 11.6% of members were dead (i.e., were not read from) and that 4.4% of object storage space was occupied by these dead data members. Usage information on member names and their types is given elsewhere (see Table 443.1 and Table 443.2).

Table 1390.1: Number of occurrences of the given token sequence. Based on the visible source of the `.c` files (`.h` files in parentheses).

Token Sequence	Occurrences	Token Sequence	Occurrences
enum {	456 (1,591)	**struct** id ;	76 (13,384)
enum id ;	0 (0)	**struct** id id	122,974 (27,589)
enum id {	474 (1,059)	**union** {	297 (725)
enum id id	2,922 (633)	**union** id ;	0 (11)
struct {	1,567 (6,503)	**union** id {	105 (2,624)
struct id {	4,407 (1,311)	**union** id id	330 (231)

bit-field maximum width

The expression that specifies the width of a bit-field shall be an integer constant expression that has a nonnegative value that shall not exceed the ~~number~~width of ~~bits in~~ an object of the type that ~~is~~would be specified ~~if~~were the colon and expression ~~are~~ omitted. 1393

struct member type

A member of a structure or union may have any object type other than a variably modified type.103) 1403

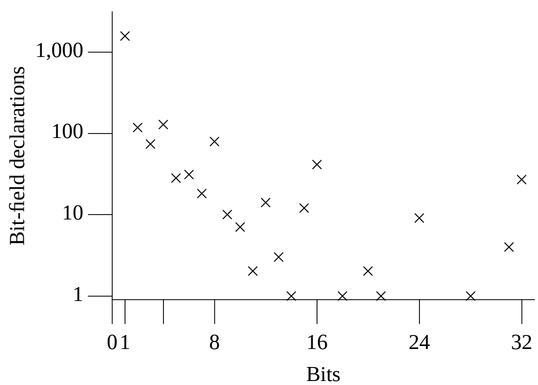


Figure 1393.1: Number of bit-field declarations specifying a given number of bits. Based on the translated form of this book's benchmark programs. (Declarations encountered in any source or header file were only counted once, the contents of system headers were ignored.)

Table 1403.1: Occurrence of structure member types (as a percentage of the types of all such members). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
int	15.8	**unsigned short**	7.7	**char** *	2.3	**void** *()	1.3
other-types	12.7	**struct**	7.2	**enum**	1.9	**float**	1.2
unsigned char	11.1	**unsigned long**	5.2	**long**	1.8	**short**	1.0
unsigned int	10.4	**unsigned**	4.0	**char**	1.8	**int** *()	1.0
struct *	8.8	**unsigned char** []	3.1	**char** []	1.5		

Table 1403.2: Occurrence of union member types (as a percentage of the types of all such members). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
struct	46.9	**unsigned int**	3.8	**double**	1.9	**char** []	1.3
other-types	11.3	**char** *	2.8	**enum**	1.7	**union** *	1.1
struct *	8.3	**unsigned long**	2.4	**unsigned char**	1.5		
int	6.0	**unsigned short**	2.1	**struct** []	1.3		
unsigned char []	4.3	**long**	2.1	(**struct** *) []	1.3		

1439 enumeration specifier syntax

```
enum-specifier:
        enum identifier_opt { enumerator-list }
        enum identifier_opt { enumerator-list , }
        enum identifier
enumerator-list:
        enumerator
        enumerator-list , enumerator
enumerator:
        enumeration-constant
        enumeration-constant = constant-expression
```

Usage

A study by Neamtiu, Foster, and Hicks[129] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 40% of releases one or more enumeration constants were added to an existing enumeration type while enumeration constants were deleted in 5% of releases and had one or more of their names changed in 16% of releases.[128]

Table 1439.1: Some properties of the set of values (the phrase *all values* refers to all the values in a particular enumeration definition) assigned to the enumeration constants in enumeration definitions. Based on the translated form of this book's benchmark programs.

Property	%
All value assigned implicitly	60.1
All values are bitwise distinct and zero is not used	8.6
One or more constants share the same value	2.9
All values are continuous , i.e. , number of enumeration constants equals maximum value minus minimum value plus 1	80.4

1463 If an identifier is provided,110) the type specifier also declares the identifier to be the tag of that type. tag declare

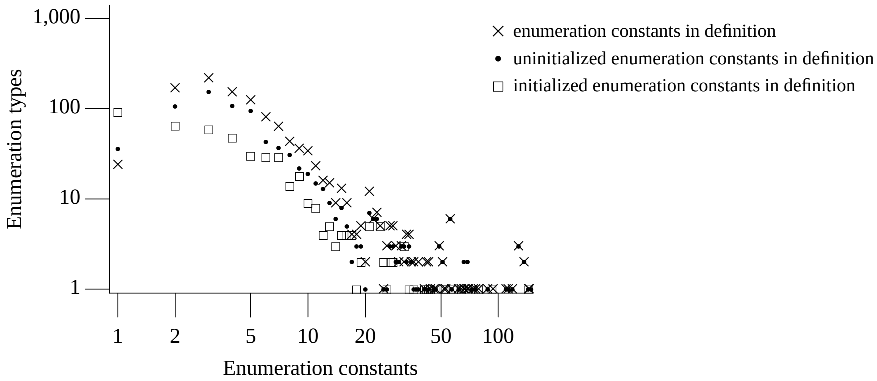


Figure 1439.1: Number of enumeration constants in an enumeration type and number whose value is explicitly or implicitly specified. Based on the translated form of this book's benchmark programs (also see Figure 298.1).

Table 1463.1: Occurrence of types declared with tag names (as a percentage of all occurrences of each keyword). Based on the visible form of the `.c` and `.h` files.

	`.c` files	`.h` files
`union` identifier	65.5	75.8
`struct` identifier	99.0	88.4
`enum` identifier	86.6	53.6

struct-or-union identifier visible

If a type specifier of the form 1472

struct-or-union identifier

or

enum *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

type qualifier syntax

1476

type-qualifier:
const
restrict
volatile

Usage

Developers do not always make full use of the **`const`** qualifier. An automated analysis[69] of programs whose declarations contained a relatively high percentage (29%) of **`const`** qualifiers found that it would have been possible to declare 70% of the declarations using this qualifier. Engblom[60] reported that for real-time embedded C code 17% of object declarations contained the **`const`** type qualifier.

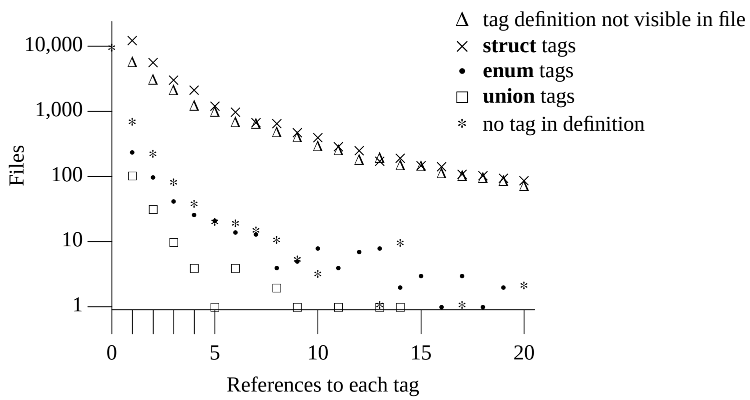


Figure 1472.1: Number of files containing a given number of references to each tag previously defined in the visible source of that file (times, bullet, square; the definition itself is not included in the count), tags with no definition visible in the `.c` file (triangle; i.e., it is defined in a header) and anonymous structure/union/enumeration definitions (star). Based on the visible form of the `.c` files.

Table 1476.1: Common token sequences containing *`type-qualifier`*s (as a percentage of each *`type-qualifier`*). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
; volatile	0.1	36.1	**{ const**	0.2	5.6
, **const**	0.2	32.8	**const unsigned**	6.2	1.4
(const	0.2	28.1	**const struct**	11.1	1.3
(volatile	0.0	26.2	**volatile unsigned**	25.6	1.1
; const	0.1	14.1	**const void**	5.3	0.8
identifier **volatile**	0.0	11.4	**volatile struct**	15.5	0.4
{ volatile	0.1	11.0	**volatile int**	7.4	0.1
const char	54.1	10.4	**volatile** identifier	36.2	0.0
static const	1.5	10.0	**volatile (**	8.9	0.0
static volatile	0.3	8.6	**const** identifier	17.6	0.0

1529 Making a function an inline function suggests that calls to the function be as fast as possible.[118)] inline suggests fast calls

Table 1529.1: Number of bytes of stack space needed by various programs before and after inlining (automatically performed by `vpcc`). *Bytes saved* refers to the amount of storage saved by optimizing the allocation of locally defined objects. Adapted from Ratliff.[142]

Program	Stack Size	Bytes Saved (%)	Inlined Stack Size	Inlined Bytes Saved (%)	Program	Stack Size	Bytes Saved (%)	Inlined Stack Size	Inlined Bytes Saved (%)
ackerman	312	8 (2.56)	232	8 (3.45)	linpack	1,504	48 (3.19)	3,312	112 (3.38)
bubblesort	568	8 (1.41)	136	8 (5.88)	mincost	1,216	0 (—)	192	8 (4.17)
cal	384	0 (—)	96	0 (—)	prof	1,584	0 (—)	400	40 (10.00)
cmp	768	0 (—)	192	0 (—)	sdiff	2,536	0 (—)	5,784	16 (0.28)
csplit	1,488	0 (—)	728	0 (—)	spline	560	8 (1.43)	200	8 (4.00)
ctags	8,144	0 (—)	24,544	88 (0.36)	tr	192	0 (—)	96	0 (—)
dhrystone	664	0 (—)	200	8 (4.00)	tsp	3,008	8 (0.27)	2,216	56 (2.53)
grep	592	0 (—)	304	0 (—)	whetstone	568	0 (—)	488	296 (60.66)
join	480	0 (—)	96	0 (—)	yacc	4,232	0 (—)	1,360	8 (0.59)
lex	9,472	0 (—)	7,208	8 (0.11)	average	1,989	4 (0.47)	2,510	34 (5.23)

1629 typedef name syntax

```
typedef-name:
        identifier
```

Usage

A study by Neamtiu, Foster, and Hicks[129] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 16% of releases one or more existing typedef names had the type they defined changed.[128]

Table 1629.1: Occurrences of types defined in a **typedef** definition (as a percentage of all types appearing in **typedef** definition). Based on the translated form of this book's benchmark programs.

Type	Occurrences	Type	Occurrences
struct	58.00	**unsigned long**	1.47
enum	9.50	**int** *()	1.46
other-types	8.86	**enum** *()	1.46
struct *	6.97	**union**	1.38
unsigned int	2.68	**long**	1.29
int	2.46	**void** *()	1.18
unsigned char	2.21	**unsigned short**	1.07

---

initialization syntax

1641

```
initializer:
        assignment-expression
        { initializer-list }
        { initializer-list , }
initializer-list:
        designation_opt initializer
        initializer-list , designation_opt initializer
designation:
        designator-list =
designator-list:
        designator
        designator-list designator
designator:
        [ constant-expression ]
        . identifier
```

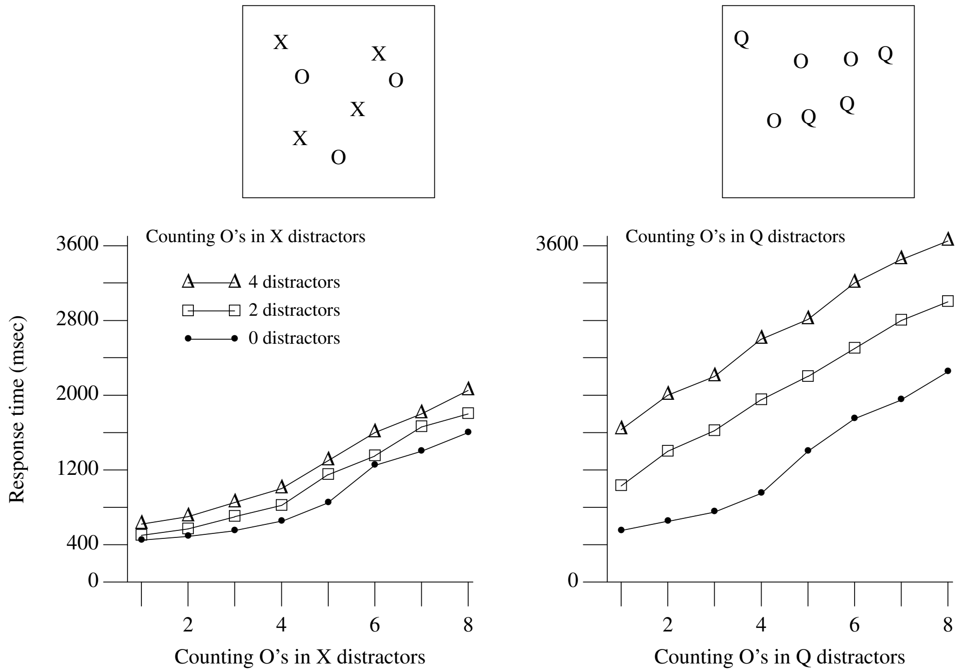


Figure 1641.1: Average time (in milliseconds) taken for subjects to enumerate O's in a background of X or Q distractors. Based on Trick and Pylyshyn.[167]

Table 1641.1: Occurrence of object types, in block scope, whose declaration includes an initializer (as a percentage of the type of all such declarations with initializers). Based on the translated form of this book's benchmark programs. Usage information on the types of all objects declared at file scope is given elsewhere (see Table 1348.2).

Type	%	Type	%
struct *	39.5	**long**	2.6
int	22.6	**char**	2.5
other-types	9.1	**unsigned short**	2.4
unsigned int	4.5	**unsigned char**	1.5
union *	4.3	**unsigned char** *	1.4
char *	4.0	**unsigned int** *	1.2
unsigned long	3.4	**enum**	1.1

Table 1641.2: Occurrence of object types with internal linkage, at file scope, whose declaration includes an initializer (as a percentage of the type of all such declarations with initializers). Based on the translated form of this book's benchmark programs. Usage information on the types of all objects declared at file scope is given elsewhere (see Table 1348.4).

Type	%	Type	%
const char []	22.5	**char** *	2.2
const struct	14.7	**int** []	2.1
int	11.1	**char** []	2.0
struct	10.4	**unsigned char** []	1.7
other-types	10.4	**void** *()	1.3
struct []	8.3	**(char *)** []	1.3
struct *	2.9	**int** *()	1.2
(const char * const) []	2.9	**const unsigned char** []	1.2
unsigned short []	2.5	**const short** []	1.2

object value indeterminate

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

statement syntax

1707

```
statement:
                labeled-statement
                compound-statement
                expression-statement
                selection-statement
                iteration-statement
                jump-statement
```

Usage

Of the approximately 2,204,000 statements in the visible form of the .c files 60.3% were *expression-statement*s, 21.3% *selection-statement*s, 15.0% *jump-statement*s, and 3.4% *iteration-statement*s. Of these 5.4% were *labeled-statement*s.

block

A *block* allows a set of declarations and statements to be grouped into one syntactic unit. 1710

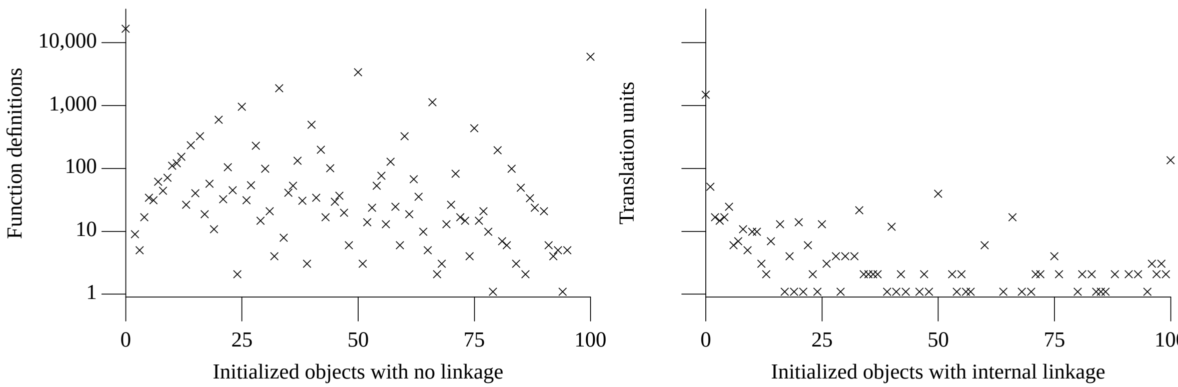


Figure 1652.1: Number of object declarations that include an initializer (as a percentage of all corresponding object declarations), either within function definitions (functions that did not contain any object definitions were not included), or within translation units and having internal linkage (while there are a number of ways of counting objects with external linkage, none seemed appropriate and no usage information is given here). Based on the translated form of this book's benchmark programs.

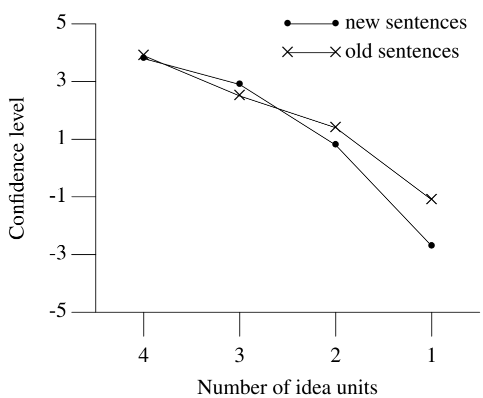


Figure 1707.1: Subject confidence level for having previously seen a sentence containing different numbers of idea units. Based on Bransford and Franks.[21]

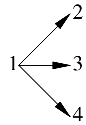
Romulus, the legendary founder of Rome, took the women of the Sabine by force.

1 (took, Romulus, women, by force)

2 (found, Romulus, Rome)

3 (legendary, Romulus)

4 (Sabine, women)



Cleopatra's downfall lay in her foolish trust in the fickle political figures of the Roman world.

1 (because, α, β)

2 $\alpha \rightarrow$ (fell down, Cleopatra)

3 $\beta \rightarrow$ (trust, Cleopatra, figures)

4 (foolish, trust)

5 (fickle, figures)

6 (political, figures)

7 (part of, figures, world)

8 (Roman, world)

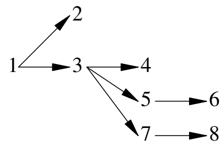


Figure 1707.2: Two sentences, one containing four and the other eight propositions, and their propositional analyses. Based on Kintsch and Keenan.[95]

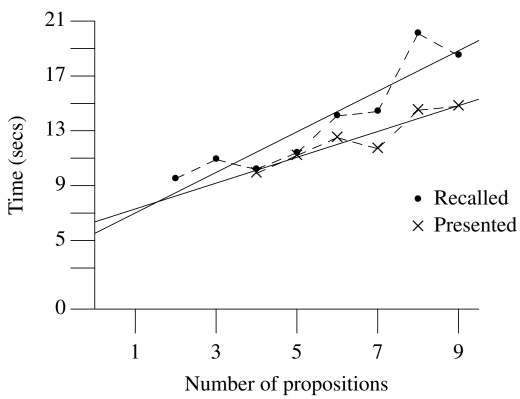


Figure 1707.3: Reading time (in seconds) and recall time for sentences containing different numbers of propositions (straight lines represent a least squares fit; for reading $t = 6.37 + .94P_{pres}$, and for recall $t = 5.53 + 1.48P_{rec}$). Adapted from Kintsch and Keenan.[95]

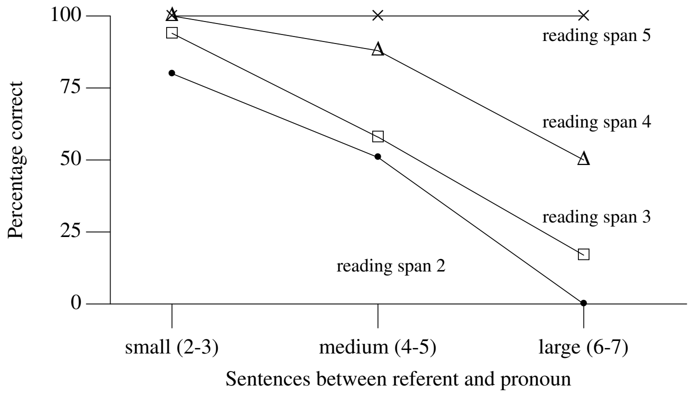


Figure 1707.4: Percentage of correct subject responses to the pronoun reference questions as a function of the number of sentences between the pronoun and the referent noun. Plotted lines are various subject reading spans. Adapted from Daneman and Carpenter.[49]

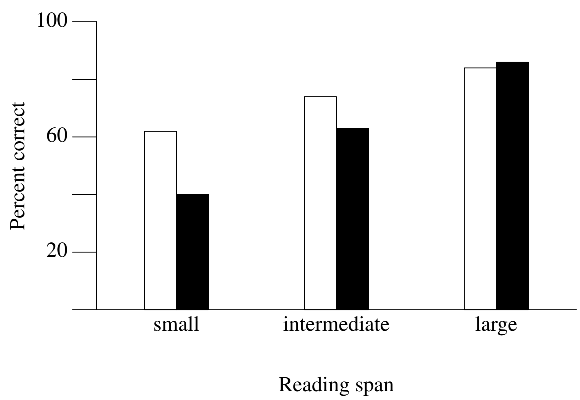


Figure 1707.5: Percentage of correct answers as a function of subject's reading span and the presence or absence of a sentence boundary. Adapted from Daneman and Carpenter.[50]

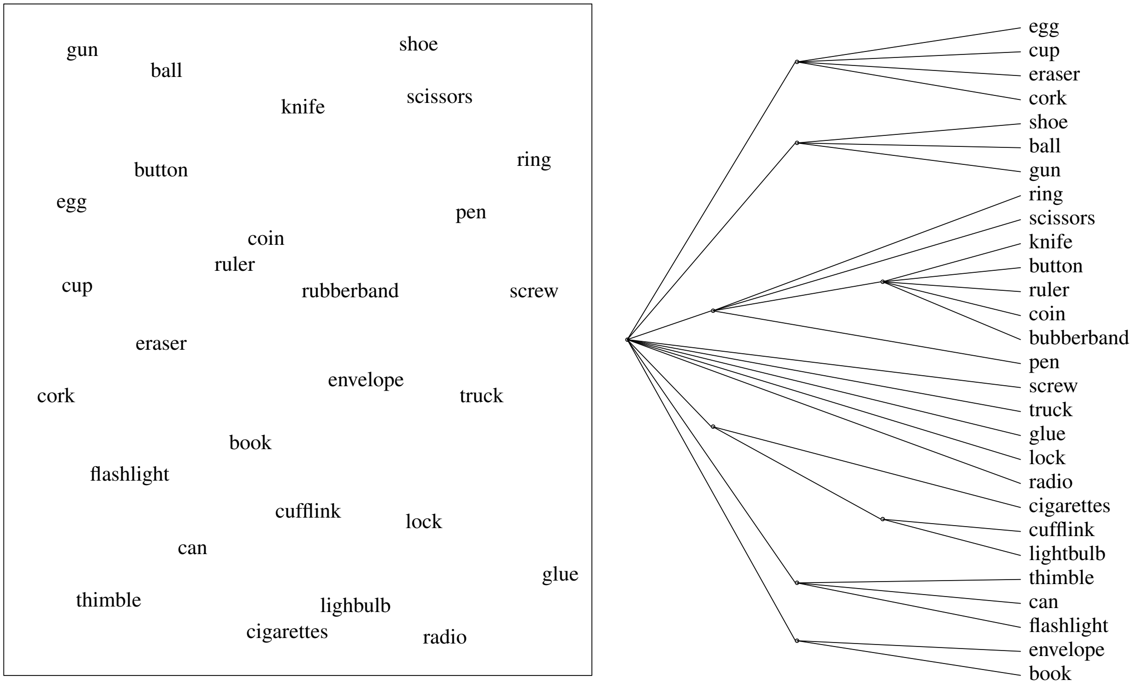


Figure 1707.6: Example of an object layout and the corresponding ordered tree for one of the subjects. Based on McNamara, Hardy, and Hirtle.[120]

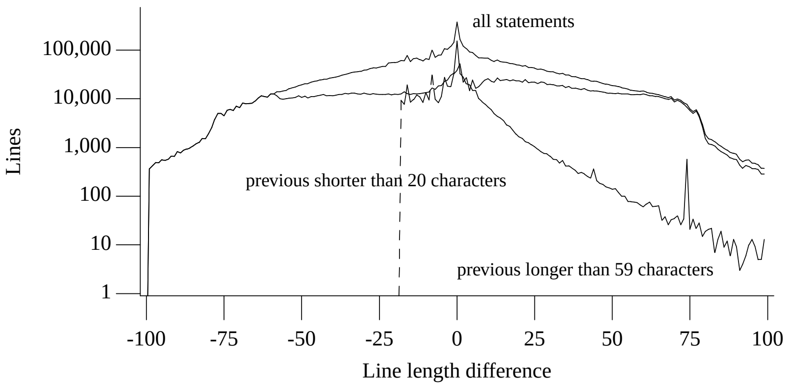


Figure 1707.7: Visible difference in offset of last non-space character on a line between successive lines, in the visible form of the `.c` files (horizontal tab characters were mapped to 8 space characters), for lines of various lengths, i.e., those whose previous line contained 60 or more characters, and those whose previous line contains less than 20 characters. There are ten times fewer lines sharing the same right offset as sharing the same left offset (see Figure 1707.8). Based on the visible form of the `.c` files.

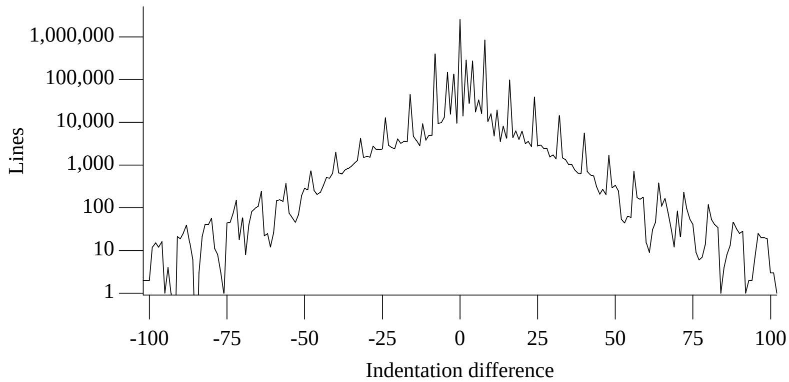


Figure 1707.8: Visible difference in relative indentation of first non-space character on a line between successive lines in the visible form of the `.c` files (horizontal tab characters were mapped to 8 space characters). The smaller peaks around zero are indentation differences of two characters. The wider spaced peaks have a separation of eight characters. Individual files had more pronounced peaks. Based on the visible form of the `.c` files.

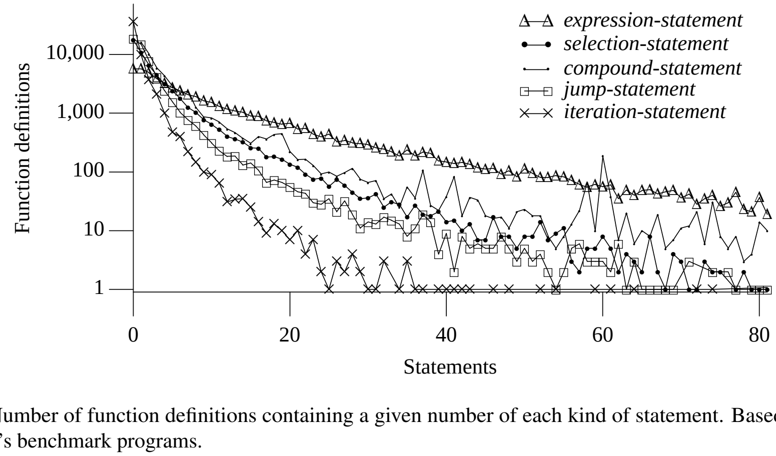


Figure 1707.9: Number of function definitions containing a given number of each kind of statement. Based on the translated form of this book's benchmark programs.

Table 1710.1: Occurrence of constructs that terminated execution of a basic block during execution of PostgreSQL processing the TPC-D benchmark. Adapted from Ramirez, Larriba-Pey, Navarro, Serrano, Torrellas, and Valero.[141]

Basic Block Type	Static Count (thousand)	Dynamic Count (billion)
Branch	54.026 (42.4%)	4.0 (50.2%)
Fall-through	31.120 (24.4%)	1.8 (22.4%)
Function return	32.052 (25.2%)	1.1 (13.7%)
Function call	10.228 (8 %)	1.1 (13.7%)

Table 1710.2: Mean number of machine instructions executed per basic block (i.e., total number of instructions executed in a function divided by the total number of basic blocks executed in that function) for a variety of SPEC benchmark programs. *Leaf* refers to functions that do not call any other functions, while *Non-Leaf* refers to functions that contain calls to other functions. Based on Calder, Grunwald, and Zorn.[27]

Program	Leaf	Non-Leaf	Program	Leaf	Non-Leaf
burg	6.8	4.9	eqntott	9.1	5.4
ditroff	6.8	4.7	espresso	5.0	5.1
tex	10.4	8.5	gcc	5.2	5.7
xfig	4.8	5.3	li	2.9	5.7
xtex	7.3	5.8	sc	3.5	4.2
compress	18.4	5.7	Mean	7.3	5.5

Usage

Usage information on block nesting is discussed elsewhere. 277 limit block nesting

1712 A *full expression* is an expression that is not part of another expression or of a declarator. full expression

1713 Each of the following is a full expression:

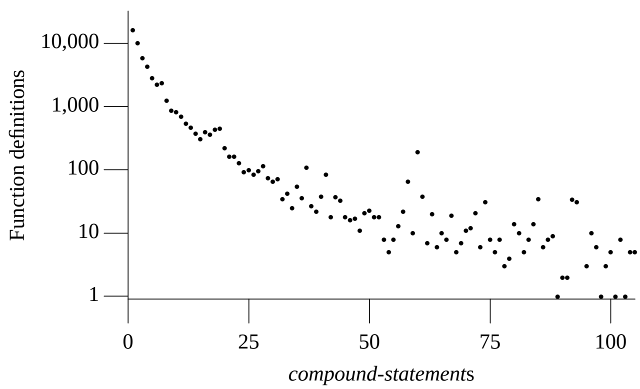


Figure 1710.1: Number of function definitions containing a given number of `compound-statement`s. Based on the translated form of this book's benchmark programs.

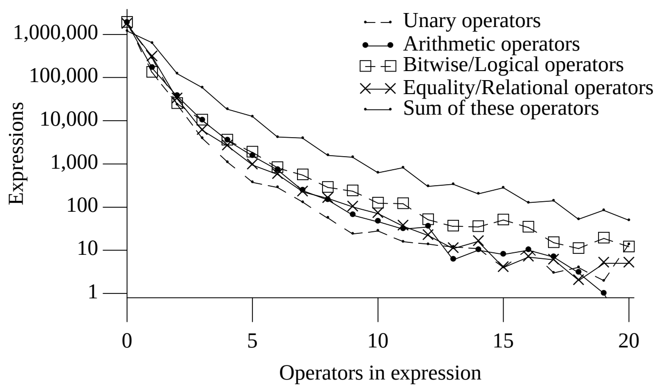


Figure 1712.1: Number of expressions containing a given number of various kinds of operator, plus a given number of all of these kinds of operators. Based on the visible form of the `.c` files.

Table 1713.1: Occurrence of full expressions in various contexts (as a percentage of all full expressions). Based on the translated form of this book's benchmark programs.

Context of Full Expression	Occurrence	Context of Full Expression	Occurrence
expression statement	65.9	**for** *expr-1*	1.6
if controlling expression	16.4	**for** controlling expression	1.5
return expression	6.2	**for** *clause-1*	1.5
object declaration initializer	4.2	**switch** controlling expression	0.6
while controlling expression	2.1		

labeled statements syntax

1722

labeled-statement:
 identifier **:** *statement*
 case *constant-expression* **:** *statement*
 default **:** *statement*

Usage

jump statement causes jump to 1783

In the translated form of this book's benchmark programs 2% of labels were not the destination of any **goto** statement. Usage information on **goto** statements is given elsewhere.

Table 1722.1: Percentage of function definitions containing a given number of labeled statements (other than a **case** or **default** label). Based on the visible form of the `.c` files.

Labels	% Functions	Labels	% Functions
1	3.5	3	0.3
2	0.9	4	0.1

case fall through

Labels in themselves do not alter the flow of control, which continues unimpeded across them. 1727

Table 1727.1: Common token pairs involving a **case** or **default** label. Based on the visible form of the .c files. Almost all of the sequences { case occur immediately after the controlling expression of the **switch** statement.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token
; default	0.4	81.4
; case	2.1	52.1
: case	15.5	22.1
{ case	2.6	15.0
} case	1.3	7.3
: default	0.5	5.7
#endif default	0.8	4.4

1729 compound statement syntax

compound-statement:
    { *block-item-list*$_{opt}$ **}**
block-item-list:
    block-item
    block-item-list block-item
block-item:
    declaration
    statement

Usage

Usage information on the number of declarations occurring in nested blocks is given elsewhere (see Figure 408.1).

1731 null statement syntax expression statement syntax

expression-statement:
    expression$_{opt}$ **;**

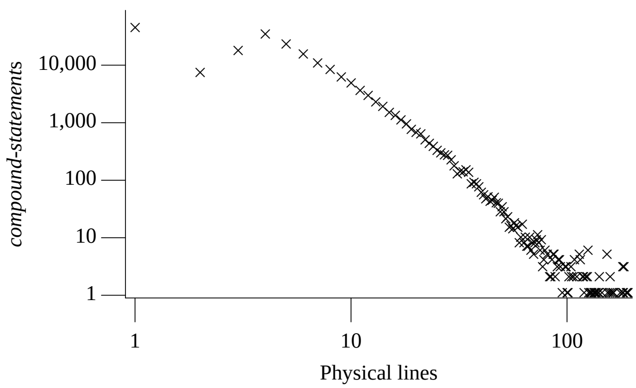


Figure 1729.1: Number of *compound-statements* containing the given number of physical lines (including the opening and closing braces and any nested *compound-statements*, but excluding the lines between the braces denoting the start/end of the function definition). Based on the translated form of this book's benchmark programs.

Table 1731.1: Occurrence of the most common forms of expression statement (as a percentage of all expression statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *`integer-constant`* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Form of *expression-statement*	%	Form of *expression-statement*	%
function-call	37	object = expression	4
object = object	16	object **v++**	2
object = function-call	10	expression	1
object = constant	7	other-expr-stmt	22

selection statement syntax

1739

selection-statement:
```
        if ( expression ) statement
        if ( expression ) statement else statement
        switch ( expression ) statement
```

Table 1739.1: Dynamic breakdown of non-loop branches for programs in SPEC89. *% of All Branches* is the percentage of all branches that are non-loop branches. *Heuristics* are the results of using the heuristics for predicting the target successor of each non-loop branch, *Perfect* the results for the perfect predictor, *Random* the results for predicting each non-loop branch randomly. *Big* is the number of non-loop branches in the program contributing more than 5% of all dynamic non-loop branches (and in parenthesis as a percentage of non-loop branches). Based on Ball and Larus.[14]

Program	% of All Branches	Heuristics	Perfect	Random	Big (%)	Program	% of All Branches	Heuristics	Perfect	Random	Big (%)
gcc	73	37	11	50	0 (0)	poly	20	40	3	31	3 (54)
lcc	71	32	12	52	1 (13)	fpppp	86	42	9	41	0 (0)
qpt	70	26	9	52	0 (0)	costScale	71	29	21	49	6 (52)
compress	66	40	18	66	6 (69)	doduc	52	33	3	49	0 (0)
xlisp	62	28	7	50	0 (0)	tomcatv	38	2	0	50	2 (98)
addalg	52	43	30	43	7 (67)	dcg	21	15	4	46	4 (51)
ghostview	52	16	4	47	4 (53)	spice2g6	21	36	8	52	2 (27)
eqntott	49	50	25	50	2 (92)	sgefat	18	26	8	61	8 (73)
rn	48	34	1	51	3 (25)	dnasa7	10	32	4	55	4 (58)
grep	44	1	0	3	3 (96)	matrix300	4	33	0	66	3 (99)
congress	40	28	3	57	2 (10)	Mean		29	10	49	
espresso	37	26	13	42	3 (24)	Std.Dev.		12	8	13	
awk	29	14	3	57	4 (29)						

Table 1739.2: Percentage of correct responses given to the four kinds of questions. Adapted from Bell and Johnson-Laird.[15]

Kind of Question	Correct 'yes' Response	Correct 'no' Response
is possible	91%	65%
is necessary	71%	81%

Table 1739.3: Percentage of subjects accepting that the stated conclusion could be logically deduced from the given premises. Based on Evans, Barston, and Pollard.[66]

Status-context	Example	Conclusion Accepted
Valid-believable		
	No Police dogs are vicious Some highly trained dogs are vicious Therefore, some highly trained dogs are not police dogs	88%
Valid-unbelievable		
	No nutritional things are inexpensive Some vitamin tablets are inexpensive Therefore, some vitamin tablets are not nutritional things	56%
Invalid-believable		
	No addictive things are inexpensive Some cigarettes are inexpensive Therefore, some addictive things are not cigarettes	72%
Invalid-unbelievable		
	No millionaires are hard workers Some rich people are hard workers Therefore, some millionaires are not rich people	13%

Table 1739.4: Properties of the two systems of thinking. Based on Stanovich.[155]

System 1	System 2
Unconscious	Conscious
Automatic	Controlled
Associative	Rule-based
Heuristic processing	Analytic processing
Undemanding of cognitive capacity	Demanding of cognitive capacity
Relatively fast	Relatively slow
Acquisition by biology, exposure, and personal experience	Acquisition by cultural and formal training
Highly contextualized	Decontextualized
Conversational and socialized	Asocial
Independent of general intelligence	Correlated with general intelligence

Table 1739.5: Eight sets of premises describing the same relative ordering between A, B, and C (peoples names were used in the study) in different ways, followed by the percentage of subjects giving the correct answer. Adapted from De Soto, London, and Handel.[52]

	Premises	Percentage Correct Response		Premises	Percentage Correct Response
1	A is better than B B is better than C	60.5	5	A is better than B C is worse than B	61.8
2	B is better than C A is better than B	52.8	6	C is worse than B A is better than B	57.0
3	B is worse than A C is worse than B	50.0	7	B is worse than A B is better than C	41.5
4	C is worse than B B is worse than A	42.5	8	B is better than C B is worse than A	38.3

Table 1739.6: Percentage *yes* responses to various forms of questions (based on 238 responses). Based on Sloman, and Lagnado.[151]

Question	Causal	Conditional
D holds?	80%	57%
A holds?	79%	36%

Table 1739.7: Occurrence of the most common conditional sentence types in speech (266 conditionals from a 63,746 word corpus) and writing (948 conditionals from 357,249 word corpus). In the notation *if + x, y*: *x* is the condition (which might, for instance, be in the past tense) and *y* can be thought of as the *then part* (which might, for instance, use one of the words would/could/might, or be in the present tense). Adapted from Celce-Murcia.[34]

Structure	Speech	Writing
If + present, present	19.2	16.5
If + present, (will/be going to)	10.9	12.5
If + past, (would/might/could)	10.2	10.0
If + present, (should/must/can/may)	9.0	12.1
If + (were/were to), (would/could/might)	8.6	6.0
If + (had/have +en), (would/could/might) have	3.8	3.3
If + present, (would/could/might)	2.6	6.1

Table 1739.8: Occurrence of various kinds of **`if`** statement controlling expressions (as a percentage of all **`if`** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *`integer-constant`* is an integer constant expression, and *expression* represents all other expressions. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%	Abstract Form of Control Expression	%
others	32.4	`!` function-call	3.8
object	15.5	object < *`integer-constant`*	2.2
object == object	8.9	object > *`integer-constant`*	1.8
`!` object	7.4	function-call == object	1.6
function-call	7.4	object > object	1.4
expression	5.7	object `!=` *`integer-constant`*	1.3
object `!=` object	4.2	function-call == *`integer-constant`*	1.2
object == *`integer-constant`*	4.0	object < object	1.1

Table 1739.9: Occurrence of various kinds of **`switch`** statement controlling expressions (as a percentage of all **`switch`** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *`integer-constant`* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
object	75.3
function-call	14.2
expression	5.2
others	3.3
`*v` object	2.0

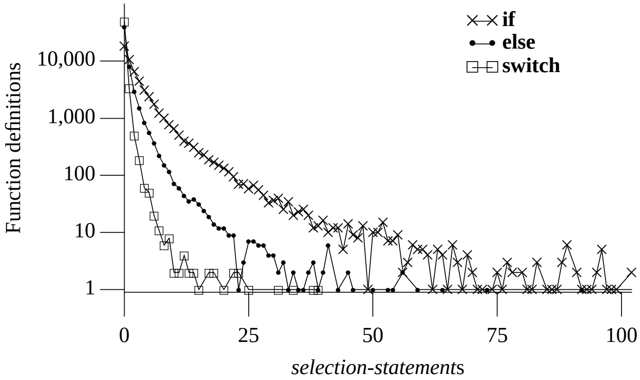


Figure 1739.1: Number of function definitions containing a given number of *selection-statement*s. Based on the translated form of this book's benchmark programs.

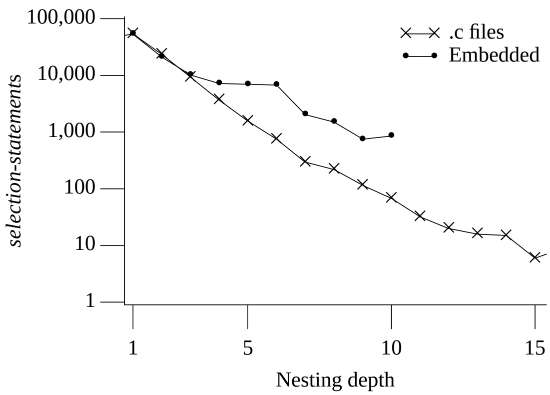


Figure 1739.2: Number of *selection-statement*s having a given maximum nesting level for embedded C[60] (whose data was multiplied by a constant to allow comparison; the data for nesting depth 5 was interpolated from the adjacent points). Based on the visible form of the `.c` files.

Table 1739.10: Occurrence of equality, relational, and logical operators in the conditional expression of an **`if`** statement (as a percentage of all such controlling expressions and as a percentage of all occurrences each operator in the source). Based on the visible form of the `.c` files. The percentage of controlling expressions may not sum to 100% because more than one of the operators occurs in the same expression.

Operator	% Controlling Expression	% Occurrence of Operator	Operator	% Controlling Expression	% Occurrence of Operator
==	31.7	88.6	>=	3.5	76.8
!=	14.1	79.7	no relational/equality	47.5	—
<	6.9	45.6	||	9.6	85.9
<=	1.9	68.6	&&	14.5	82.3
>	3.5	84.9	no logical operators	84.2	—

1740 A selection statement selects among a set of statements depending on the value of a controlling expression. controlling expression if statement

Usage

In the translated form of this book's benchmark programs 1.3% of *selection-statement*s and 4% of *iteration-statement*s have a controlling expression that is a constant expression. Use of simple, non-iterative, flow analysis enables a further 0.6% of all controlling expressions to be evaluated to a constant expression at translation time.

---

else

In the **else** form, the second substatement is executed if the expression compares equal to 0. 1745

Usage

In the visible form of the .c files 21.5% of **if** statements have an **else** form. (Counting all forms of *if* supported by the preprocessor, with **#elif** counting as both an *if* and an *else*, there is an **#else** form in 25.0% of cases.)

---

switch statement

The controlling expression of a **switch** statement shall have integer type. 1748

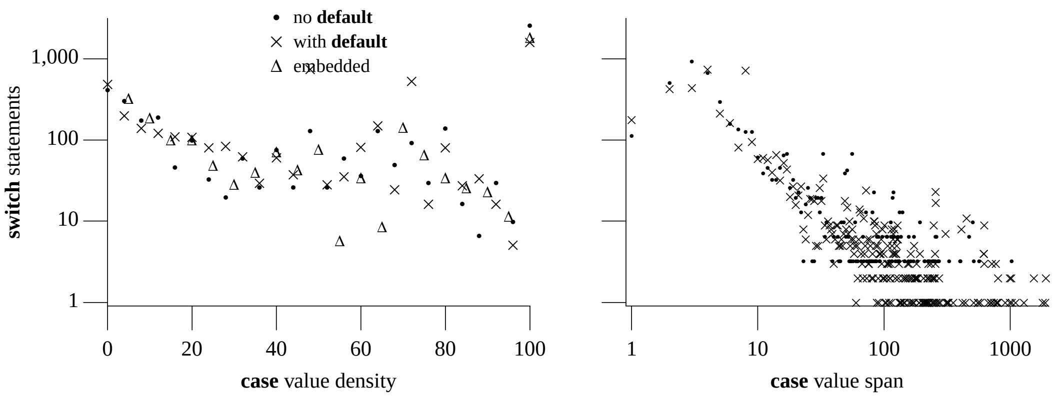


Figure 1748.1: *Density* of **case** label values (calculated as (maximum **case** label value minus minimum **case** label value minus one) divided by the number of **case** labels associated with a **switch** statement) and span of **case** label values (calculated as (maximum **case** label value minus minimum **case** label value minus one)). Based on the translated form of this book's benchmark programs and embedded results from Engblom[60] (which were scaled, i.e., multiplied by a constant, to allow comparison). The *no* ***default*** results were scaled so that the total count of **switch** statements matched those that included a **default** label.

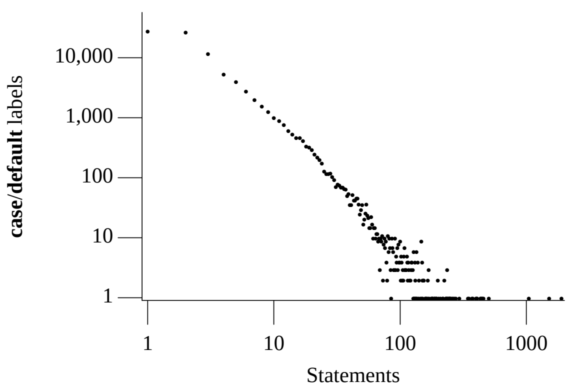


Figure 1748.2: Number of **case/default** labels having s given number of statements following them (statements from any nested **switch** statements did not contribute towards the count of a label). Based on the visible form of the .c files.

Table 1748.1: Occurrence of **switch** statements having a controlling expression of the given type (as a percentage of all **switch** statements). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	29.5	bit-field	3.1
unsigned long	18.7	**unsigned short**	2.8
enum	14.6	**short**	2.5
unsigned char	12.4	**long**	0.9
unsigned int	10.0	other-types	0.2
char	5.1		

1751 There may be at most one **default** label in a **switch** statement. default label at most one

Usage

In the visible form of the .c files, 72.8% of **switch** statements contain a **default** label.

1753 A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. switch statement causes jump

Table 1753.1: Performance comparison (in seconds) of some implementation techniques for a series of **if** statements (contained in a loop that iterated 10,000,000 times) using (1) linear search (LS), or (2) indirect jump (IJ), for a variety of processors in the SPARC family. *br* is the average number of branches per loop iteration. Based on Uh and Whalley.[170]

Processor Implementation	2.5br LS	4.5br LS	8.5br LS	2.5br IJ	4.5br IJ	8.5br IJ
SPARCstation-IPC	3.82	5.53	8.82	2.61	2.71	2.76
SPARCstation-5	1.03	1.65	2.74	0.63	0.76	0.76
SPARCstation-20	0.93	1.60	2.65	0.87	0.93	0.94
UltraSPARC-1	0.50	1.16	1.56	1.50	1.51	1.51

1763 iteration statement syntax

```
iteration-statement:
        while ( expression ) statement
        do statement while ( expression ) ;
        for ( expression_opt ; expression_opt ; expression_opt ) statement
        for ( declaration expression_opt ; expression_opt ) statement
```

Usage

A study by Bodík, Gupta, and Soffa[19] found that 11.3% of the expressions in SPEC95 were loop invariant.

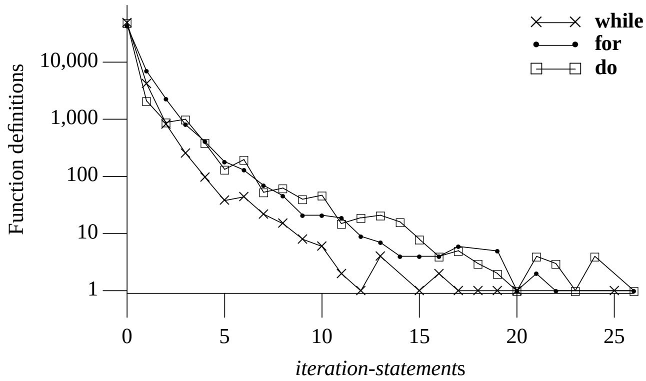


Figure 1763.1: Number of function definitions containing a given number of `iteration-statement`s. Based on the translated form of this book's benchmark programs.

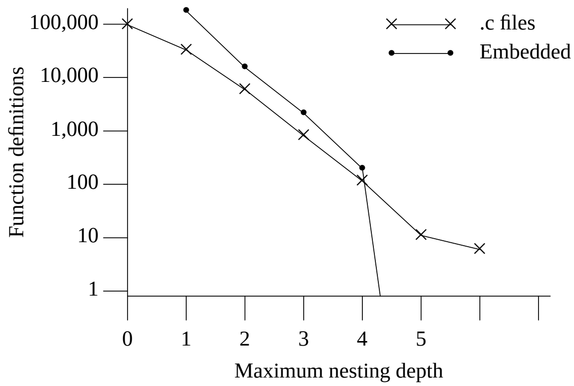


Figure 1763.2: Number of functions containing `iteration-statement`s nested to the given maximum nesting level; for embedded C[60] (whose data was multiplied by a constant to allow comparison) and the visible form of the `.c` files (zero nesting depth denotes functions not containing any `iteration-statement`s).

Table 1763.1: Occurrence of various kinds of **`for`** statement controlling expressions (as a percentage of all such expressions). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`); *assignment* is an assignment expression, `integer-constant` is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of **`for`** loop header	%
assignment **;** identifier < identifier **;** identifier **v++**	33.2
assignment **;** identifier < `integer-constant` **;** identifier **v++**	11.3
assignment **;** identifier **;** assignment	7.0
assignment **;** identifier < expression **;** identifier **v++**	3.3
assignment **;** identifier < identifier **;** **++v** identifier	2.7
; ;	2.5
assignment **;** identifier **!=** identifier **;** assignment	2.5
assignment **;** identifier <= identifier **;** identifier **v++**	2.2
assignment **;** identifier >= `integer-constant` **;** identifier **v--**	1.6
assignment **;** identifier < function-call **;** identifier **v++**	1.4
assignment **;** identifier < identifier **;** identifier **v++** , identifier **v++**	1.4
others	31.1

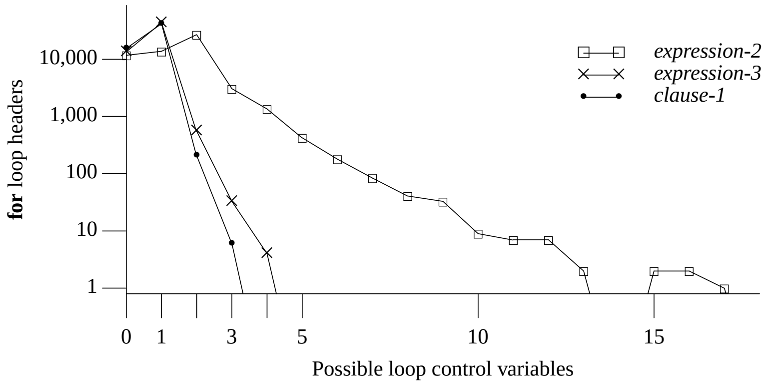


Figure 1774.1: Number of possible loop control variables appearing in *expression-2* (square-box) after filtering against the objects appearing in *expression-3* (cross) and after filtering against the objects appearing in *clause-1* (bullet). Based on the visible form of the `.c` files.

Table 1763.2: Occurrence of various kinds of **`while`** statement controlling expressions (as a percentage of all **`while`** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`); *assignment* is an assignment expression, *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%	Abstract Form of Control Expression	%
others	43.5	expression	2.2
object	12.2	*v object	2.0
object != object	7.0	assignment	1.8
integer-constant	6.2	! object	1.6
object < object	4.7	! function-call	1.3
function-call	4.4	object != *integer-constant*	1.2
object > *integer-constant*	4.0	object v-- > *integer-constant*	1.1
object v--	3.2	! expression	1.0
assignment != object	2.4		

1774 The statement for statement

for (*clause-1* ; *expression-2* ; *expression-3*) *statement*

behaves as follows:

Table 1774.1: Occurrence of sequences of components omitted from a **`for`** statement header (as a percentage of all **`for`** statements). Based on the visible form of the `.c` files.

Components Omitted	%
clause-1	3.8
clause-1 expr-2	0.1
clause-1 expr-2 expr-3	2.5
clause-1 expr-3	0.1
expr-2	0.8
expr-2 expr-3	0.2
expr-3	1.6

1782 jump statement syntax

jump-statement:

```
goto identifier ;
continue ;

break ;
return expressionopt ;
```

Usage

Numbers such as those given in Table 1782.1 and Table 1782.2 depend on the optimizations performed by an implementation. For instance, unrolling a frequently executed loop will reduce the percentage of branch instructions.

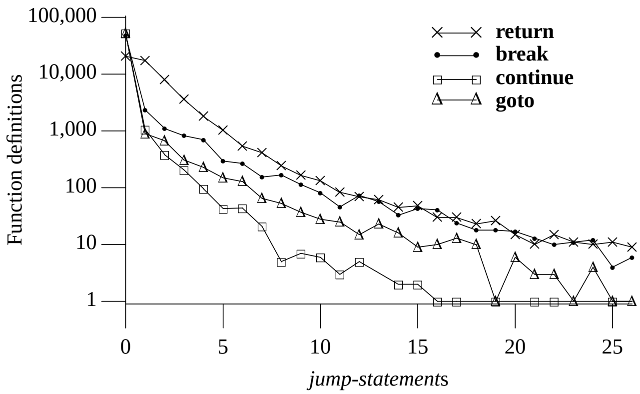


Figure 1782.1: Number of function definitions containing a given number of *jump-statement*s. Based on the translated form of this book's benchmark programs.

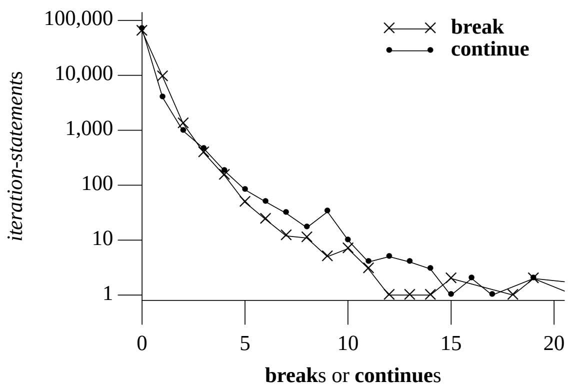


Figure 1782.2: Number of *iteration-statement* containing the given number of **break** and **continue** Based on the visible form of the .c files.

Table 1782.1: Dynamic occurrence of different kinds of instructions that can change the flow of control. *%Instructions Altering Control Flow* is expressed as a percentage of all executed instructions. All but the last row are expressed as percentages of these, control flow altering, instructions only. The kinds of instructions that change control flow are: conditional branches *CB*, unconditional branches *UB*, indirect procedure calls *IC*, procedure calls *PC*, procedure returns *Ret*, and other breaks *Oth* (e.g., signals and **switch** statements). *Instructions between branches* is the mean number of instructions between conditional branches. Based on Calder, Grunwald, and Zorn.[27]

Program	%Instructions Altering Control Flow	%CB	%UB	%IC	%PC	%Ret	%Oth	%Conditional Branch Taken	Instructions Between Branches
burg	17.1	74.1	6.9	0.0	9.5	9.5	0.0	68.8	7.9
ditroff	17.5	76.3	4.2	0.1	9.7	9.8	0.0	58.1	7.5
tex	10.0	75.9	10.7	0.0	5.8	5.8	1.9	57.5	13.2
xfig	17.5	73.6	7.7	0.6	8.6	9.2	0.3	54.8	7.8
xtex	14.1	78.2	8.5	0.2	6.0	6.2	1.0	53.3	9.1
compress	13.9	88.5	7.6	0.0	2.0	2.0	0.0	68.3	8.1
eqntott	11.5	93.5	2.1	1.5	0.7	2.2	0.0	90.3	9.3
espresso	17.1	93.2	1.9	0.1	2.3	2.4	0.1	61.9	6.3
gcc	16.0	78.9	7.4	0.4	6.1	6.5	0.8	59.4	7.9
li	17.7	63.9	8.7	0.4	12.9	13.2	0.9	49.3	8.9
sc	22.3	83.5	3.9	0.0	6.3	6.3	0.0	64.3	5.4
Mean	15.9	80.0	6.3	0.3	6.3	6.6	0.5	62.4	8.3

Table 1782.2: Number of static conditional branches sites that are responsible for the given quantile percentage of dynamically executed conditional branches. For instance, 19 conditional branch sites are responsible for over 50% of the dynamically executed branches executed by burg. *Static count* is the total number of conditional branch instructions in the program image. Of the 17,565 static branch sites, 69 branches account for the execution of 50% of all dynamic conditional branches. Not all branches will be executed during each program execution because many branches are only encountered during error conditions, or may reside in unreachable or unused code. Based on Calder, Grunwald, and Zorn.[27]

Program	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	99%	100%	Static count
burg	1	3	5	9	19	33	58	95	135	162	268	859	1,766
ditroff	3	11	19	28	38	50	64	91	132	201	359	867	1,974
tex	3	7	15	26	39	58	89	139	259	416	788	2,369	6,050
xfig	8	31	74	138	230	356	538	814	1,441	2,060	3,352	7,476	25,224
xtex	2	8	15	22	36	63	104	225	644	1,187	2,647	6,325	21,597
compress	1	2	2	3	4	5	6	8	12	14	16	230	1,124
eqntott	1	1	1	2	2	2	2	3	14	42	72	466	1,536
espresso	4	10	19	30	44	63	88	121	163	221	470	1,737	4,568
gcc	13	38	77	143	245	405	641	991	1,612	2,309	3,724	7,639	16,294
li	2	4	7	11	16	22	29	38	52	80	128	557	2,428
sc	2	3	4	6	9	16	30	47	76	135	353	1,465	4,478
Mean	3	10	21	38	62	97	149	233	412	620	1,107	2,726	7,912

1783 A jump statement causes an unconditional jump to another place. jump statement causes jump to

Usage

A study by on Gellerich, Kosiol, and Ploedereder[72] analyzed **goto** usage in Ada and C. In the translated form of this book's benchmark programs 20.6% of **goto** statements jumped to a label that occurred textually before them in the source code.

1800 A **return** statement without an expression shall only appear in a function whose return type is **void**. return without expression

Usage

The translated form of this book's benchmark programs contained 19 instances of a **return** statement without an expression appearing in a function whose return type was **void**.

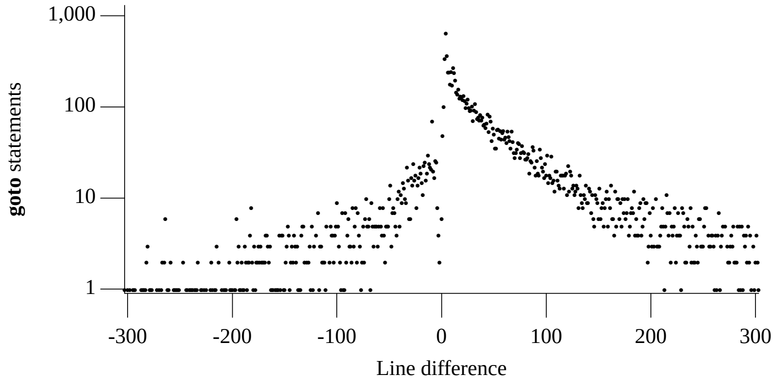


Figure 1783.1: Number of **goto** statements having a given number of visible source lines between a **goto** statement and its destination label (negative values denote backward jumps). Based on the translated form of this book's benchmark programs.

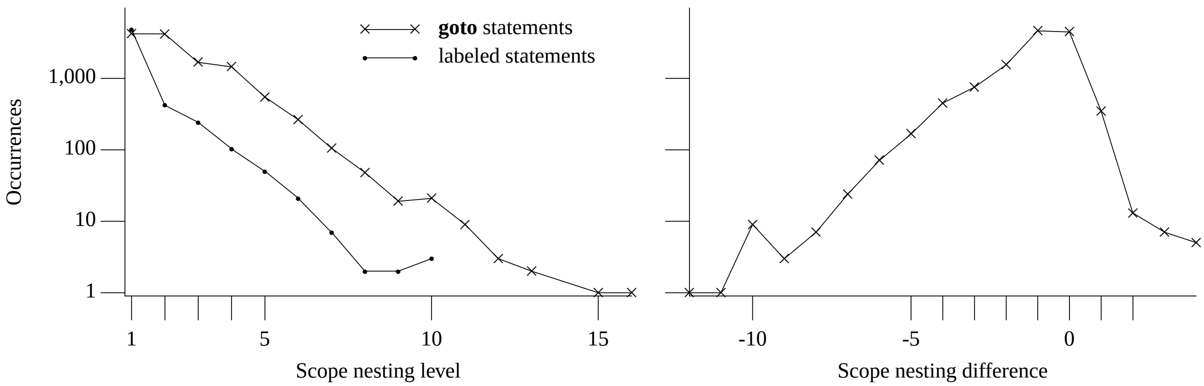


Figure 1783.2: Number of **goto** statements and labels having a given scope nesting level (nesting level 1 is the outermost block of a function definition), and on the right the difference in scope levels between a **goto** statement and its corresponding labeled statement (negative values denote a jump to a less nested scope). Based on the translated form of this book's benchmark programs.

translation unit syntax
external declaration syntax

1810

translation-unit:
  external-declaration
  translation-unit external-declaration
external-declaration:
  function-definition
  declaration

Usage

duplicate code
latent semantic analysis

On a large development project it is possible that more than one person will write some set of functions performing similar operations. This duplication of functionality occurs at a higher-level than copying and reusing sequences of statements (discussed elsewhere), it is a concept that is being duplicated. Marcus and Maletic[116] used latent semantic analysis to identify related source files (what they called *concept clones*). Source code identifiers and words in comments were used as input to the indexing process. An analysis of the Mozilla source code highlighted two different implementations of linked list functions and four files that contained their own implementations.

function definition syntax

1821

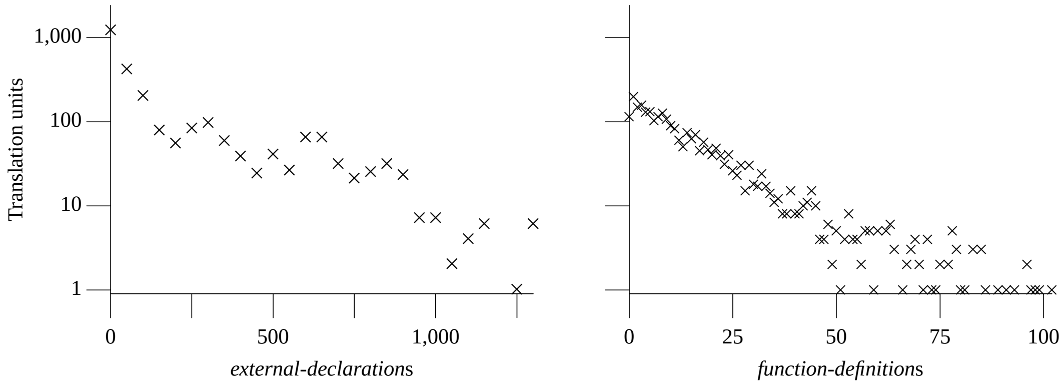


Figure 1810.1: Number of translation units containing a given number of *external-declarations* and *function-definitions* declarations (rounded to the nearest fifty and excluding identifiers declared in any system headers that are **#include**d). Based on the translated form of this book's benchmark programs.

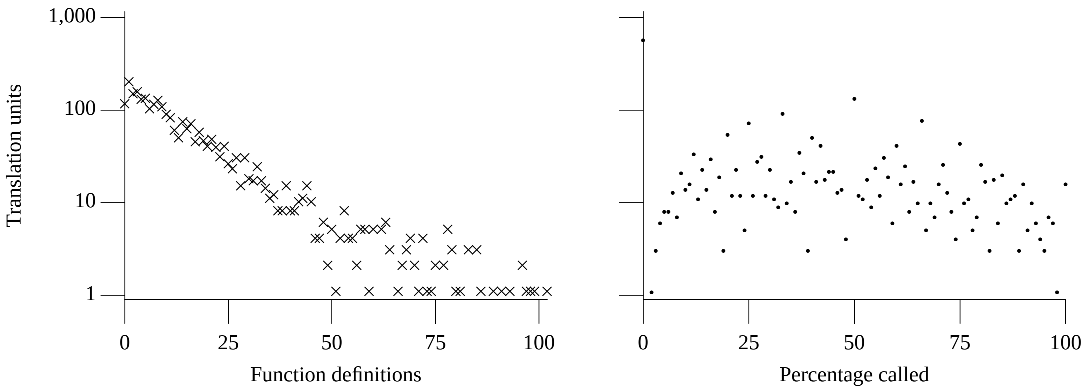


Figure 1810.2: Number of translation units containing a given number of function definitions and percentage of functions that are called within the translation unit that defines them. Based on the translated form of this book's benchmark programs.

function-definition:

declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

Table 1821.1: Probability of subjects recalling or recognizing typical or atypical actions present in stories read to them, at two time intervals (30 minutes and 1 week) after hearing them. Based on Graesser, Woll, Kowalski, and Smith.[73]

Memory Test	Typical (30 mins)	Atypical (30 mins)	Typical (1 week)	Atypical (1 week)
Recall (correct)	0.34	0.32	0.21	0.04
Recall (incorrect)	0.17	0.00	0.15	0.00
Recognition (correct)	0.79	0.79	0.80	0.60
Recognition (incorrect)	0.59	0.11	0.69	0.26

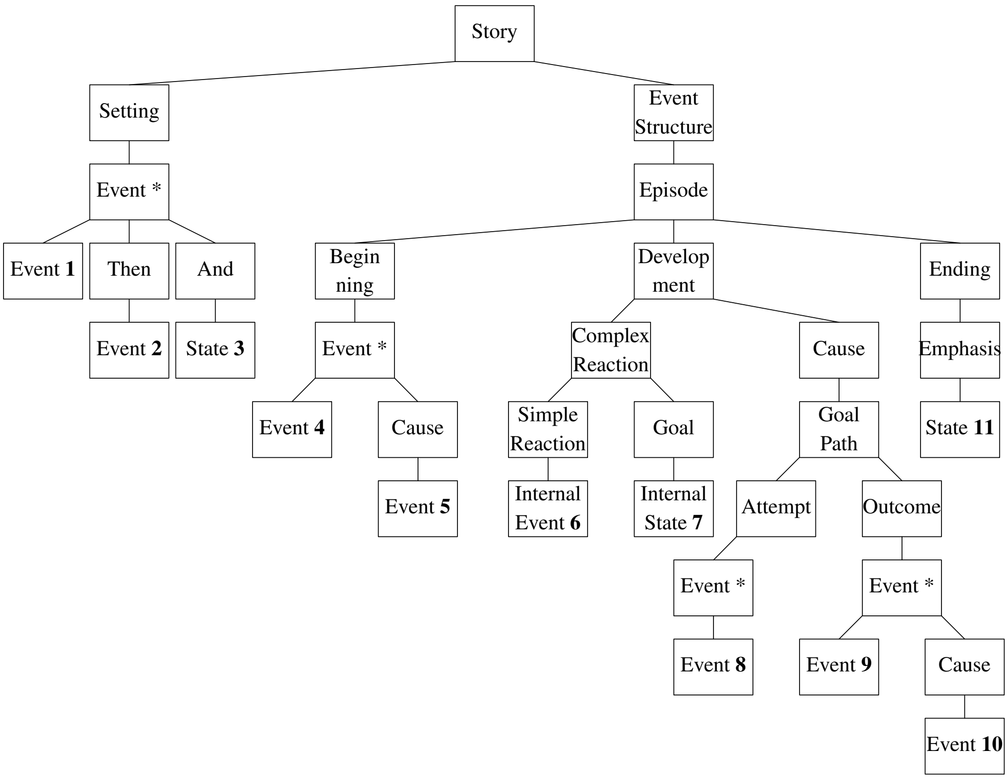


Figure 1821.1: Parse, using the Story grammar, of the tale of a dog and piece of meat. Adapted from Mandler and Johnson.[115]

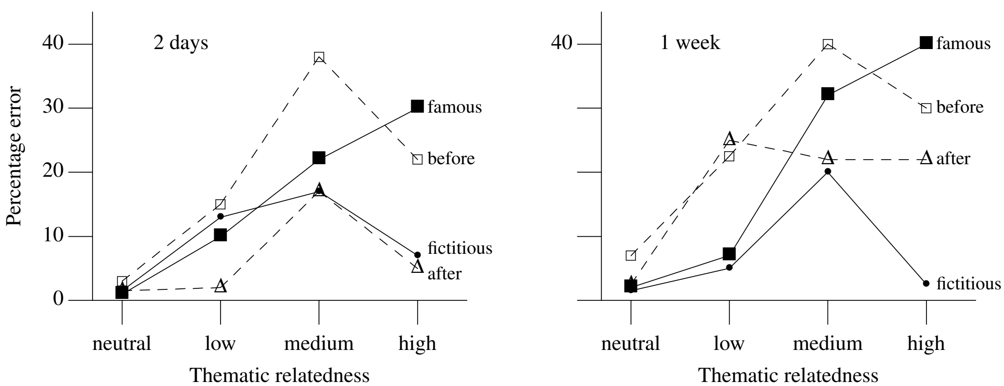


Figure 1821.2: Percentage of false-positive recognition errors for biographies having varying degrees of thematic relatedness to the famous person, in *before*, *after*, *famous*, and *fictitious* groups. Based on Dooling and Christiaansen.[57]

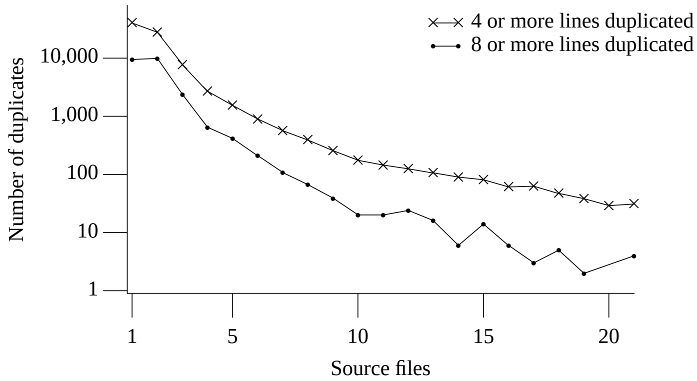


Figure 1821.3: Number of instances of duplicate physical lines, where a given duplicate line sequence is contained within a single source file or more than one source file (ignoring comments and blank lines) for sequences having at least 4 and 8 lines. Data created by processing the `.c` files (for each of the book's program's complete source tree) using `Simian`.[144]

Table 1821.2: Number of clones (the same sequence of 30 or more tokens, with all identifiers treated as equivalent) detected by CCFinder between three different operating systems (Linux, FreeBSD, and NetBSD). Adapted from Kamiya, Kusumoto, and Inoue.[93]

O/S pairs	Number of Clone Pairs	% of Lines Included in a Clone	% of Files Containing a Clone
FreeBSD/Linux	1,091	FreeBSD (0.8) Linux (0.9)	FreeBSD (3.1) Linux (4.6)
FreeBSD/NetBSD	25,621	FreeBSD (18.6) NetBSD (15.2)	FreeBSD (40.1) NetBSD (36.1)
Linux/NetBSD	1,000	Linux (0.6) NetBSD (0.6)	Linux (3.3) NetBSD (2.1)

Usage

A study of over 3,000 C functions by Harrold, Jones, and Rothermel[79] found that the size of a functions control dependency graph was linear in the number of statements (the theoretical worst-case is quadratic in the number of statements).

A study by Neamtiu, Foster, and Hicks[129] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 81% of releases one or more existing function definitions had their argument signature changed, while one or more function definitions had their return type changed in 42% of releases and one or more function definitions had their name changed in 49% of releases.[128]

Table 1821.3: Static count of number of functions and uncalled functions in SPECint95. Adapted from Cheng.[37]

Benchmark	Lines of Code	Number of Functions	Uncalled Functions	Benchmark	Lines of Code	Number of Functions	Uncalled Functions
008.espresso	14,838	361	46	126.gcc	205,583	2,019	187
023.eqntott	12,053	62	2	130.li	7,597	357	1
072.sc	8,639	179	8	132.ijpeg	29,290	477	16
085.cc1	90,857	1,452	51	134.perl	26,874	276	13
124.m88ksim	19,092	252	13	147.vortex	67,205	923	295

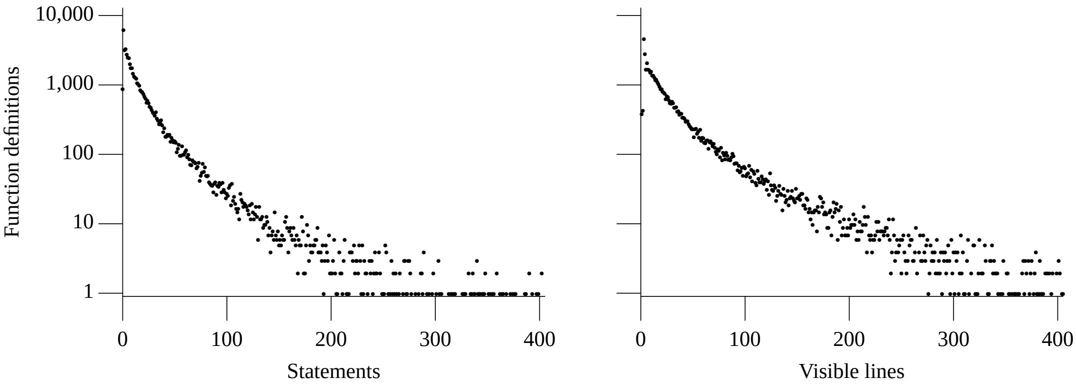


Figure 1821.4: Number of function definitions containing a given number of statements and visible source lines. Based on the translated form of this book's benchmark programs.

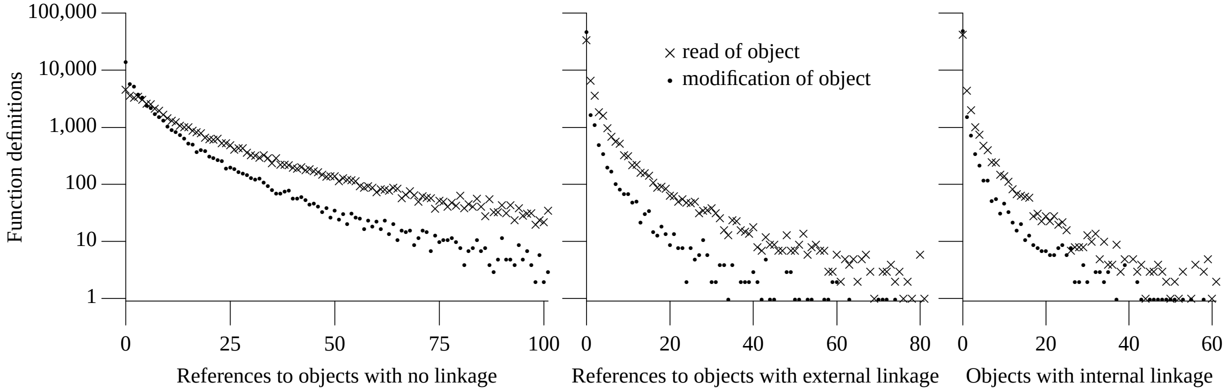


Figure 1821.5: Number of function definitions containing a given number of references (i.e., an access or modification) to all objects, having various kinds of linkage. Based on the translated form of this book's benchmark programs.

Table 1821.4: Mean number of instructions executed per function invocation. Based on Calder, Grunwald, and Zorn.[27]

Program	Mean	Leaf	Non-Leaf	Program	Mean	Leaf	Non-Leaf
burg	61.6	30.6	142.8	eqntott	386.8	402.8	294.2
ditroff	58.6	72.3	56.3	espresso	244.9	151.3	526.5
tex	173.2	44.3	205.4	gcc	96.4	30.1	123.5
xfig	61.9	38.6	74.8	li	42.5	31.9	44.2
xtex	114.9	93.9	136.5	sc	71.1	49.4	80.1
compress	368.4	1,360.2	367.5	Mean	152.8	209.6	186.5

Table 1821.5: Contents of function bodies (as a percentage of all bodies) for embedded `.c` source,[61] SPECint95, and the translated form of this book's benchmark programs.

	Embedded	SPECint95	Book benchmarks
Trivial (one basic block)	32.7	16.2	57.1
Non-looping	47.9	48.1	18.1
Looping	19.4	35.7	24.8

function definition return type

The return type of a function shall be **`void`** or an object type other than array type. 1823

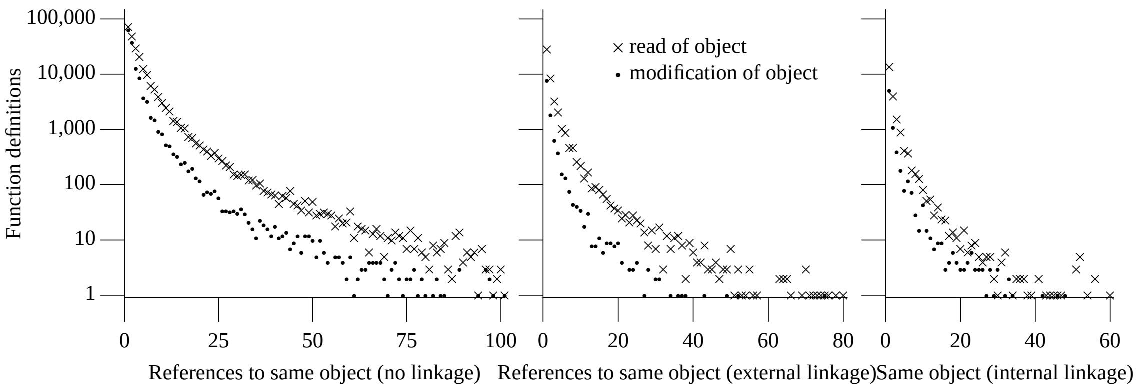


Figure 1821.6: Number of function definitions containing a given number of references (i.e., an access or modification) to the same object, having various kinds of linkage. Based on the translated form of this book's benchmark programs.

Usage

Usage information on function return types in the `.c` files is given elsewhere (see Table 1005.1).

Table 1823.1: Occurrence of function return types (as a percentage of all return types; signedness and number of bits appearing in value representation form) appearing in the source of embedded applications (5,597 function definitions) and the SPECint95 benchmark (2,713 function definitions). A likely explanation of the greater use of type **void** is the perceived performance issues associated with returning values via the stack causing developers to return values via objects at file scope. Adapted from Engblom.[61]

Type/Representation	Embedded	SPECint95	Type/Representation	Embedded	SPECint95
void	59.4	31.2	ptr-to . . .	2.0	17.1
unsigned 32 bit	0.5	2.2	signed 32 bit	0.3	48.4
unsigned 16 bit	3.3	0.0	signed 16 bit	1.6	0.2
unsigned 8 bit	31.6	0.5	signed 8 bit	0.8	0.0

1831 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters.

Usage

Information on argument types is given elsewhere (see Table 1003.1).

Table 1831.1: Occurrence of parameter types in function definitions (as a percentage of the parameters in all function definitions). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
struct *	44.4	**void ***	3.4	**long**	1.6	**struct * ***	1.2
int	14.7	**union ***	3.1	**int ***	1.5	**enum**	1.2
other-types	6.8	**unsigned long**	2.7	**unsigned char ***	1.4	**const char ***	1.1
unsigned int	5.1	**unsigned int ***	2.0	**char * ***	1.3	**long ***	1.0
char *	4.7	**unsigned char**	1.6	**unsigned short**	1.2		

1844 If the **}** that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined. function termination reaching }

Usage

In the translated source of this book's benchmark programs 0.7% of function definitions contained both `return;` (or the flow of control reached the terminating **}**) and `return expr;`.

preprocessor directives syntax

1854

preprocessing-file:
  group$_{opt}$
group:
  group-part
  group group-part
group-part:
  if-section
  control-line
  text-line
  # *non-directive*
if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*
if-group:
  # **if** *constant-expression new-line group*$_{opt}$
  # **ifdef** *identifier new-line group*$_{opt}$
  # **ifndef** *identifier new-line group*$_{opt}$
elif-groups:
  elif-group
  elif-groups elif-group
elif-group:
  # **elif** *constant-expression new-line group*$_{opt}$
else-group:
  # **else** *new-line group*$_{opt}$
endif-line:
  # **endif** *new-line*
control-line:
  # **include** *pp-tokens new-line*
  # **define** *identifier replacement-list new-line*
  # **define** *identifier lparen identifier-list*$_{opt}$ **)**
      replacement-list new-line
  # **define** *identifier lparen* **...)** *replacement-list new-line*
  # **define** *identifier lparen identifier-list* **, ...)**
      replacement-list new-line
  # **undef** *identifier new-line*
  # **line** *pp-tokens new-line*
  # **error** *pp-tokens*$_{opt}$ *new-line*
  # **pragma** *pp-tokens*$_{opt}$ *new-line*
  # *new-line*
text-line:
  pp-tokens$_{opt}$ *new-line*
non-directive:
  pp-tokens new-line
lparen:
  a **(** character not immediately preceded by white-space
replacement-list:
  pp-tokens$_{opt}$
pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token
new-line:
  the new-line character

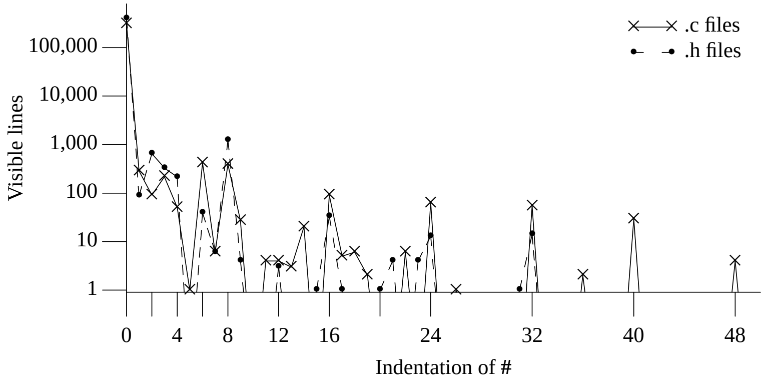


Figure 1854.1: Number of lines containing a preprocessing directive starting at a given indentation from the start of the line (i.e., amount of white space before the first # on a line, with the tab character treated as eight space characters). Based on the visible form of `.c` and `.h` files.

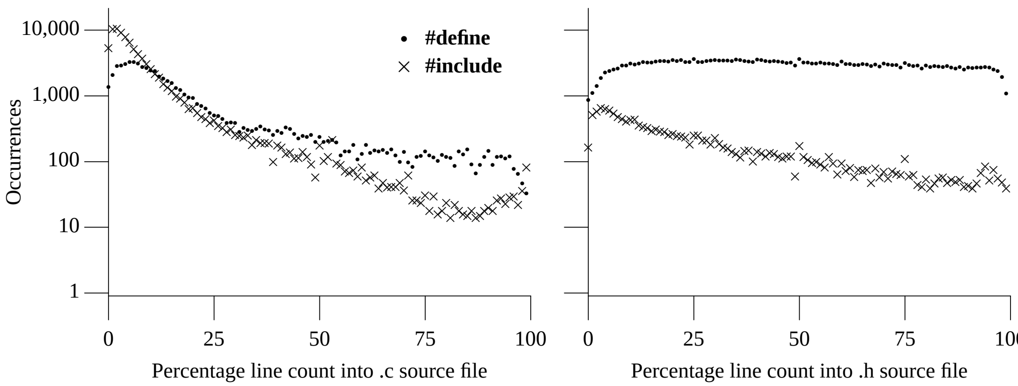


Figure 1854.2: Number of **`#include`** and **`#define`** directives appearing at a relative location (i.e., 100*line_number/lines_in_file) in the source. Based on the visible form of `.c` and `.h` files.

Usage

A study by Ernst, Badros, and Notkin[63] provides one of the few empirical studies of C preprocessor use.

Table 1854.1: Occurrence of preprocessor directive names and preprocessor operators (as a percentage of all directive names and operators). Based on the visible form of the `.c` and `.h` files.

Directive Name	`.c` file	`.h` file	Directive Name	`.c` file	`.h` file
`#define`	19.9	75.0	**`#if`**	6.2	1.5
`#endif`	19.9	7.2	**`##`**	0.3	0.9
`#include`	28.6	4.1	**`#elif`**	0.2	0.2
`#ifndef`	2.4	3.2	**`#pragma`**	0.1	0.1
`#ifdef`	11.3	2.5	**`#error`**	0.2	0.1
`#else`	4.8	1.7	**`#`**	0.0	0.1
defined	3.6	1.7	**`#line`**	1.4	0.0
`#undef`	1.0	1.6			

1872 and it may contain unary operator expressions of the form #if defined

```
defined identifier
```

or

```
defined ( identifier )
```

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

Table 1872.1: Occurrence of controlling expressions containing the **defined** operator (as a percentage of all **#if** and **#elif** preprocessing directives). The **#elif** preprocessing directive was followed by the **defined** operator in 66.5% of occurrences of that preprocessing directive— in the .c files (.h 75.5%). Based on the visible form of the .c and .h files.

Preprocessing Directive	%
#if defined (identifier)	15.7
#if defined (identifier) || defined (identifier)	5.8
#if defined (identifier) && defined (identifier)	2.0
#if ! defined (identifier)	1.9
#elif defined (identifier)	1.9
#if defined (identifier) && ! defined (identifier)	1.3
#if ! defined (identifier) && ! defined (identifier)	0.9
#if defined (identifier) || defined (identifier) || defined (identifier)	0.8
#if defined identifier || defined identifier	0.5
#if ! defined (identifier) && ! defined (identifier) && ! defined (identifier)	0.3
others	5.3

Preprocessing directives of the forms 1875

```
# if   constant-expression new-line group_opt
# elif constant-expression new-line group_opt
```

check whether the controlling constant expression evaluates to nonzero.

Usage

The visible form of the .c files contained 12,277 (.h 4,159) **#else** directives.

Table 1875.1: Common **#if** preprocessing directive controlling expressions (as a percentage of all **#if** directives). Where *integer-constant* is an integer constant expression, and *function-call* is an invocation of a function-like macro. Based on the visible form of the .c files.

Abstract Form of Control Expression	%
identifier	26.5
integer-constant	20.3
defined (identifier)	16.4
defined (identifier) || defined (identifier)	6.0
identifier == identifier	2.4
identifier > *integer-constant*	2.4
identifier >= function-call	2.1
defined (identifier) && defined (identifier)	2.0
! defined (identifier)	2.0
defined (identifier) && ! defined (identifier)	1.3
identifier >= *integer-constant*	1.3
identifier != *integer-constant*	1.1
identifier < function-call	1.1
! identifier	1.1
others	14.0

Table 1875.2: Common **#elif** preprocessing directive controlling expressions (as a percentage of all **#elif** directives). Where *integer-constant* is an integer constant expression, and *function-call* is a function-like macro. Based on the visible form of the .c files.

Abstract Form of Control Expression	%
defined (identifier)	49.7
identifier == identifier	19.4
defined identifier	6.6
defined (identifier) || defined (identifier)	5.7
identifier	4.7
defined (identifier) && defined (identifier)	2.6
identifier == *integer-constant*	1.9
identifier >= function-call	1.2
defined (identifier) || defined (identifier) || defined (identifier)	1.2
identifier >= *integer-constant*	1.0
others	6.1

1878 After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. #if identifier replaced by 0

Usage

Approximately 15% of all conditional inclusion directives, in the translated form of this book's benchmark programs, contained an identifier that was replaced by 0 (i.e., they contained an identifier that was neither the operand of **defined** or defined as macro names).

1889 If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals;

1896 A **#include** directive shall identify a header or source file that can be processed by the implementation. source file inclusion

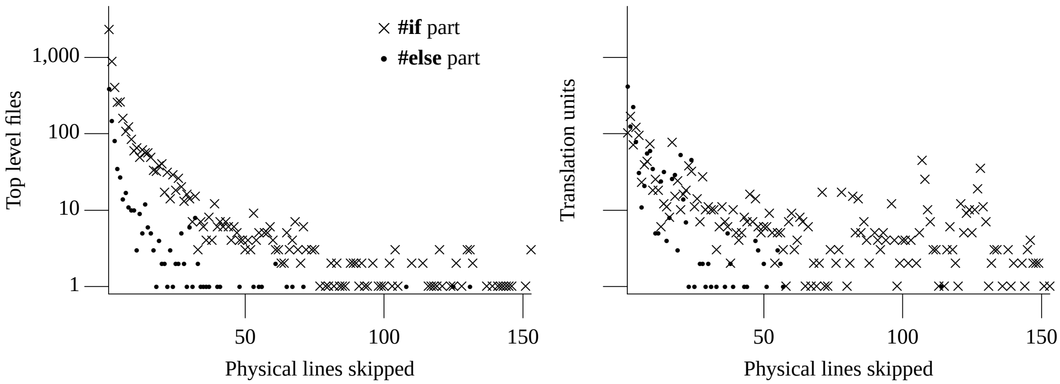


Figure 1889.1: Number of top-level source files (i.e., the contents of any included files are not counted) and (right) complete translation units (including the contents of any files **#include**d more than once) having a given number of lines skipped during translation of this book's benchmark programs.

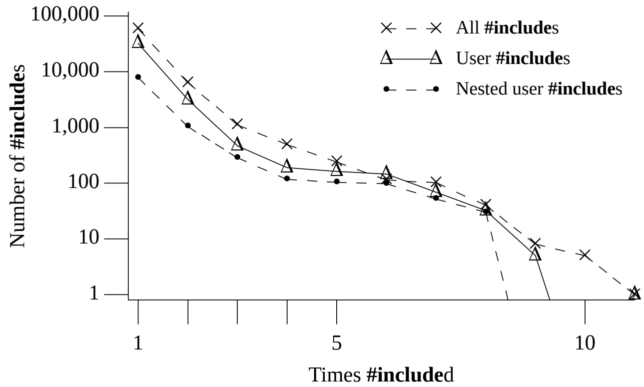


Figure 1896.1: Number of times the same header was **`#include`**d during the translation of a single translation unit. The crosses denote all headers (i.e., all systems headers are counted), triangles denote all headers delimited by quotes (i.e., likely to be user defined headers) and bullets denote all quote delimited headers **`#include`** nested at least three levels deep. Based on the translated form of this book's benchmark programs.

Figure 1896.2: Number of preprocessing translation units (excluding system headers) containing a given number of **`#include`**s whose contents are not referenced during translation (excludes the case where the same header is **`#include`**d more than once, see Figure 1896.1). Based on the translated form of this book's benchmark programs.

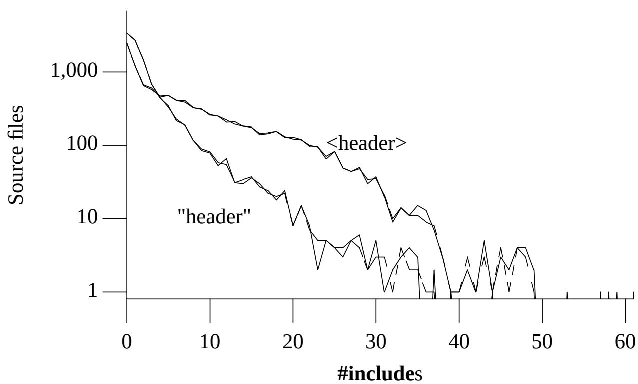


Figure 1896.3: Number of `.c` source files containing a given number of **`#include`** directives (dashed lines represent number of unique headers). Based on the visible form of the `.c` files.

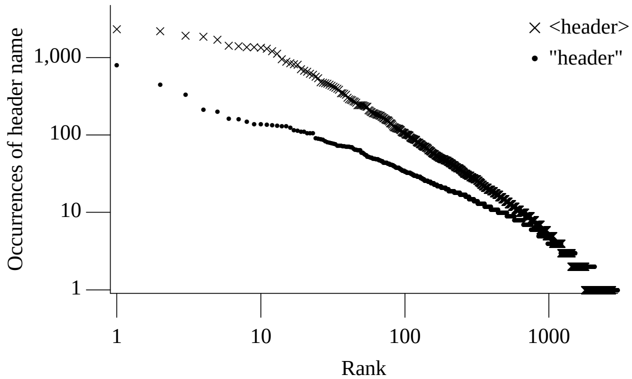


Figure 1896.4: *header-name* rank (based on character sequences appearing in **#include** directives) plotted against the number of occurrences of each character sequence. Also see Figure 792.26. Based on the visible form of the .c files.

Table 1896.1: Occurrence of two forms of *header-name*s (as a percentage of all **#include** directives), the percentage of each kind that specifies a path to the header file, and number of absolute paths specified. Based on the visible form of the .c files.

Header Form	% Occurrence	% Uses Path	Number Absolute Paths
<h-char-sequence>	75.0	86.4	0
"q-char-sequence"	25.0	17.2	0

1897 A preprocessing directive of the form #include h-char-sequence

 # include *<h-char-sequence> new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header.

Table 1897.1: Number of various kinds of identifiers declared in the headers contained in the /usr/include directory of some translation environments. Information was automatically extracted and represents an approximate lower bound. Versions of the translation environments from approximately the same year (mid 1990s) were used. The counts for ISO C assumes that the minimum set of required identifiers are declared and excludes the type generic macros.

Information	Linux 2.0	AIX on RS/6000	HP/UX 9	SunOS 4	Solaris 2	ISO C
Number of headers	2,006	1,514	1,264	987	1,495	24
macro definitions	10,252	18,637	13,314	11,987	10,903	446
identifiers with external linkage	1,672	1,542	1,935	616	1,281	487
identifiers with internal linkage	80	34	2012	0	5	0
tag declaration	716	1,088	899	1,208	945	3
typedef name declared	1,024	828	15	493	1,027	55

1931 A preprocessing directive of the form macro object-like

 # define *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name[146)] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.

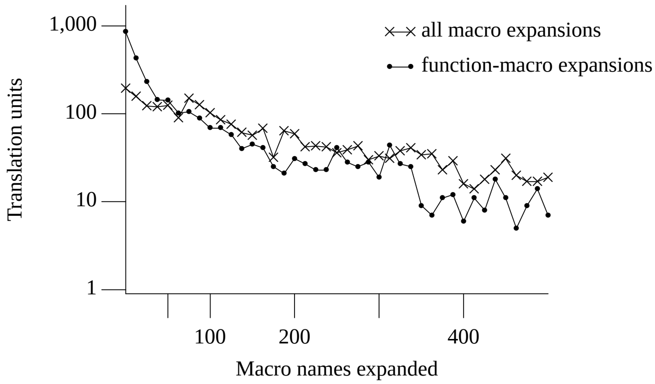


Figure 1931.1: Number of translation units containing a given number of macro names which were macro expanded, excluding expansions that occurred while processing the contents of system headers. Based on the translated form of this book's benchmark programs.

Usage

limit 287 macro definitions

Usage information on the number of macro names defined in source files is given elsewhere.

Table 1931.1: Detailed breakdown of the kinds of replacement lists occurring in macro definitions. Adapted from Ernst, Badros, and Notkin.[63]

Replacement List	%	Example
constant	42	`#define ARG_MAX 1000`
expression	33	`#define SHFT_UP(x) ((x) << 8)`
empty	6.9	`#define DUMMY`
unknown identifier	6.9	`#define INTERN_BUF buffer`
statement	5.1	`#define TERMINATE goto func_end`
type	2.1	`#define NODE_PTR void *`
other	1.9	`#define OPTION -X=23`
symbol	1.4	`#define ALLOC_STORAGE malloc`
syntactic	0.5	`#define begin {`

Table 1931.2: Common macro definitions listed with an abstracted form of their replacement list (as a percentage of all macro definitions). Note that *function-call* may also be a macro invocation. Based on the visible form of the `.c` and `.h` files.

Kind of Macro Defined and Abstract Form of its Replacement List	%
object-like macro *integer-constant*	50.7
object-like macro identifier	5.9
object-like macro expression	5.8
function-like macro function-call	4.7
object-like macro function-call	3.7
object-like macro *string-literal*	3.4
function-like macro expression	3.4
object-like macro	3.4
object-like macro *constant-expression*	2.0
function-like macro	1.7
others	15.4

macro parameter scope extends

The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **`#define`** preprocessing directive. 1934

Usage

Usage information on the number of parameters in function-like macro definitions is given elsewhere. 290 limit macro parameters

---

1950 Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list. # operator

Usage

Based on the visible form of the `.c` files 0.26% (0.09% `.h` files) of the replacement lists of macro definitions contained a **#** operator. There were no obvious patterns to the usage.

1958 A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition. ## operator

Table 1958.1: Occurrence of the **##** preprocessor operator (as a percentage of all occurrences of that operator). The form `, ## identifier` is a `gcc` extension (described in the Common implementations subclause). Based on the visible form of the `.c` and `.h` files.

Preprocessing Token Sequence	%
identifier ## identifier	70.2
, ## identifier	24.2
identifier ## identifier ## identifier	15.7
others	4.8
integer-constant ## identifier	1.8
integer-constant ## identifier ## *integer-constant*	1.0
identifier ## *integer-constant*	1.0

1961 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token.

Table 1961.1: Possible results of concatenating, using the **##** operator, pairs of preprocessing tokens (the one appearing in the left column followed by the one appearing in the top row) where the result might be defined (*undefined* denotes undefined behavior).

	identifier	*pp-number*	*punctuator*	*string-literal*	*character-constant*
identifier	identifier	identifier or undefined	undefined	string-literal or undefined	character-constant or undefined
pp-number	pp-number	pp-number	pp-number or undefined	undefined	undefined
punctuator	pp-number or undefined	pp-number or undefined	punctuator or undefined	undefined	undefined
everything else	undefined	undefined	undefined	undefined	undefined

---

1976 A preprocessing directive of the form #undef

```
# undef identifier new-line
```

causes the specified identifier no longer to be defined as a macro name.

Usage

Approximate 5% of all **#undef** directives occur before a **#include** directive (based on the visible form of the `.c` files).

Table 1976.1: Occurrence of various sequences of preprocessing directives (as a percentage of all such sequences) that follow a **#undef** and reference the same identifier (e.g., 2.7% of the first occurrence of **#undef** are followed by one or more **#define**s followed by one or more **#undef**s). **#define** represents one or more **#define** preprocessing directives. **#undef** represents one or more **#undef** preprocessing directives. **#if[n]def** represents two or more **#if**s and **#ifndef**s, in any order. **#und-def** represents one or more pairs of **#undef #define** preprocessing directives. Based on the visible form of the `.c` files.

Following Directive Sequences	%
	53.0
#ifdef	20.4
#define	16.2
others	4.8
#define #undef	2.7
#if(n)def	1.5
#define #undef-#define **#undef**	1.4

macro name predefined

The following macro names[151)] shall be defined by the implementation: 2004

Table 2004.1: Occurrence of predefined macro names (as a percentage of all predefined macro names; a total of 1,826). Based on the visible form of the `.c` and `.h` files.

Predefined Macro	`.c` files	`.h` files	Predefined Macro	`.c` files	`.h` files	Predefined Macro	`.c` files	`.h` files
__LINE__	42.17	43.47	__TIME__	2.52	0.00	__STDC_IEC_559__	0.00	0.0
__FILE__	36.31	37.77	__STDC_VERSION__	0.00	0.00	__STDC_HOSTED__	0.00	0.0
__STDC__	15.77	18.11	__STDC_ISO_10646__	0.00	0.00			
__DATE__	3.23	0.65	__STDC_IEC_559_COMPLEX__	0.00	0.00			

identifier linkage future language directions

Declaring an identifier with internal linkage at file scope without the **static** storage-class specifier is an obsolescent feature. 2035

Usage

The translated form of this book's benchmark programs contained 12 declarations of an identifier with internal linkage at file scope without the **static** storage-class specifier.

2044

Table 2044.1: Occurrence of calls to C library functions (as a percentage of all calls). Based on the translated form of this book's benchmark programs.

Function	%	Function	%	Function	%	Function	%
fprintf	1.468	memmove	0.093	strstr	0.028	ferror	0.016
sprintf	0.978	fclose	0.085	sin	0.028	atof	0.016
printf	0.902	strchr	0.077	mblen	0.028	strncat	0.015
strlen	0.824	fopen	0.077	realloc	0.026	ftell	0.015
strcmp	0.730	fabs	0.065	memcmp	0.021	tolower	0.014
strcpy	0.533	signal	0.045	fputs	0.021	fscanf	0.014
free	0.397	getenv	0.045	strerror	0.020	abort	0.014
memcpy	0.324	abs	0.044	cos	0.020	qsort	0.013
memset	0.321	perror	0.040	strtok	0.019	mbtowc	0.013
exit	0.218	fwrite	0.034	strrchr	0.019	fseek	0.013
malloc	0.201	fflush	0.034	sqrt	0.019	calloc	0.013
strncmp	0.194	sscanf	0.032	ungetc	0.018	mbstowcs	0.012
strcat	0.190	vsprintf	0.031	floor	0.017	feof	0.012
rand	0.179	fread	0.030	ceil	0.017	atol	0.012
strncpy	0.145	snprintf	0.029	toupper	0.016	wcstombs	0.011
atoi	0.110	time	0.028	fgets	0.016		

Table 2044.2: Percentage of instructions executed in developer written code and implementation libraries (main library `libc`, and maths library `libm`) of C programs that do not use the X11 libraries. Based on Calder, Grunwald, and Srivastava.[26]

Programs	main	libc	libm	libots	libcurses	Programs	main	libc	libm	libots	libcurses
alvinn	97.25	2.12	0.63			li	99.71	0.29			
compress	99.98	0.02				m88ksim	99.75	0.03	—	0.22	
ditroff	87.80	12.20				perl	70.70	29.30			
ear	90.33	6.12	3.55			sc	53.03	18.42	—	—	28.55
eqntott	94.29	5.71				vortex	95.11	4.89			
espresso	93.93	6.07				Mean	90.15	7.10	0.35	0.02	2.38
go	99.99	0.01									

Table 2044.3: Percentage of instructions executed in developer written code and implementation libraries of C programs that use the X11 libraries. Based on Calder, Grunwald, and Srivastava.[26]

Programs	main	libc	libm	libX11	libXaw	libXext	libXm	libXmu	libXt
cbzone	48.10	11.80	7.60	32.14	—				0.36
ghostview	3.38	23.39	—	20.93	7.53	0.02		0.08	44.68
gs	91.88	4.99	0.18	2.93	—				0.02
xanim	62.40	29.96	0.06	4.36	0.09				3.13
xfig	4.95	15.05	0.15	28.58	9.84			0.14	41.30
xkeycaps	6.47	18.45		43.15	3.70	0.01		0.06	28.15
xmgr	22.95	12.13	0.04	23.24	—		17.05	—	24.60
xpaint	14.11	11.01	—	25.43	0.77			0.02	48.66
xpilot	68.64	24.24	0.03	7.09	—				
xpool	53.17	0.26	44.91	1.65	—				
xtex	45.02	23.86	—	23.09	2.95			0.03	5.05
xv	74.07	25.46	0.01	0.46	—				
Mean	41.26	16.72	4.41	17.75	2.07	0.00	1.42	0.03	16.33

2063 The standard headers are
<assert.h> <inttypes.h> <signal.h> <stdlib.h> <complex.h> <iso646.h> <stdarg.h> <string.h> <ctype.h> <limits.h> <stdbool.h> <tgmath.h> <errno.h> <locale.h> <stddef.h> <time.h> <fenv.h> <math.h> <stdint.h> <wchar.h> <float.h> <setjmp.h> <stdio.h> <wctype.h>

Table 2063.1: Number of standard headers appearing in a **#include** directive. Based on the visible form of the `.c` and `.h` files.

Header name	.c file	.h file	Header name	.c file	.h file
stdio.h	1,424	175	signal.h	213	10
stdlib.h	860	100	locale.h	23	7
stddef.h	107	90	stdint.h	0	3
string.h	828	83	inttypes.h	1	1
errno.h	481	82	float.h	25	1
setjmp.h	81	80	wctype.h	1	0
stdarg.h	167	54	wchar.h	2	0
time.h	185	47	tgmath.h	0	0
ctype.h	291	35	stdbool.h	0	0
limits.h	88	32	iso646.h	0	0
assert.h	91	26	fenv.h	1	0
math.h	246	21	complex.h	0	0

2175 The header <ctype.h>declares several functions useful for classifying and mapping characters.166) In all cases the argument is an int, the value of which shall be representable as an unsigned char or shall equal the value of the macro EOF. If the argument has any other value, the behavior is undefined. ctype.h header

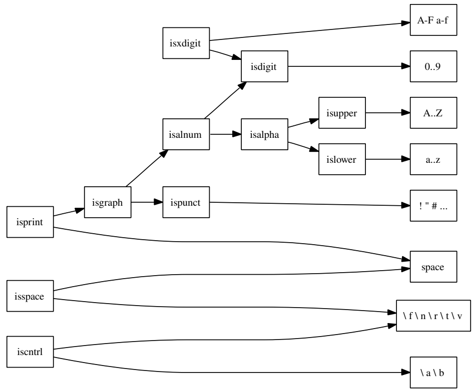


Figure 2175.1: Interrelationships between the character handling functions.

Additional macro definitions, beginning with E and a digit or E and an uppercase letter,171) may also be 2203
specified by the implementation.

Usage

errno.h future directions

See Future Library Directions for reserved identifier usage information.

Each of the macros 2216
FE_DOWNWARD FE_TONEAREST FE_TOWARDZERO FE_UPWARD
is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the fegetround and fesetround functions. Additional implementation-defined rounding directions, with macro definitions beginning with FE_ and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.174)

Usage

Usage information on reserved identifier spellings is given elsewhere (see Table 98.1).

Additional implementation-defined environments, with macro definitions beginning with FE_ and an uppercase 2221
letter, and having type "pointer to const-qualified fenv_t", may also be specified by the implementation.

Usage

Usage information on reserved identifier spellings is given elsewhere (see Table 98.1).

---

math.h header

The header **<math.h>** declares two types and many mathematical functions and defines several macros. Most 2320
synopses specify a family of functions consisting of a principal function with one or more double parameters, a double return value, or both; and other functions with the same name but with f and l suffixes, which are corresponding functions with float and long double parameters, return values, or both.189) Integer arithmetic functions and conversion functions are discussed later.

Usage

A study by Citron and Feitelson[40] (based on the SPEC CFP95, Khoros, and MediaBench suites) found that on average 82% of calls to functions in the maths library, by a program, had the same argument values as previous calls to those function, by the same program.

An implementation need not generate any of these signals, except as a result of explicit calls to the raise 2508
function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters SIG and an uppercase letter or with SIG_ and an uppercase letter,210) may also be

specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

Usage

See Future Library Directions for reserved identifier usage information. signal.h future directions

2590 The header <stdio.h>declares three types, several macros, and many functions for performing input and output. stdio.h header

Usage

A study Pasquale and Polyzos[134] looked at the I/O characteristics of scientific applications, a study by Hsu, Smith, and Young[86] measured I/O behavior for production database workloads, while Ruemmler and Wilkes[146] looked at disk access patterns.

2839 The order and contiguity of storage allocated by successive calls to the `calloc`, `malloc`, and `realloc` functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object. memory management functions

Table 2839.1: Memory management function usage. *mallocs* is the number of calls to the `malloc` library function, *frees* the number of calls to the `free` library function, and *size* the mean number of bytes of the objects allocated. Based on Calder, Grunwald, and Zorn.[27]

Program	mallocs	frees	size	Program	mallocs	frees	size
burg	23,098	2,895	843.4	eqntott	85	0	23,981.6
ditroff	0	0	—	espresso	190,386	190,077	122.5
tex	60	32	1,727.1	gcc	1,043	903	1,353.4
xfig	7,260	4,070	193.6	li	27	0	3,407.5
xtex	2,944	1,131	358.9	sc	6,985	2,419	52.0
compress	1	0	16.0				

2885 div function ldiv function lldiv function

Table 2885.1: Table 7.2: Results of div, ldiv and lldiv

numer	denom	quot	rem
7	3	2	1
-7	3	-2	-1
7	-3	-2	1
-7	-3	2	-1

References

1. P. L. Ackerman and E. D. Heggestad. Intelligence, personality, and interests: Evidence for overlapping traits. *Psychological Bulletin*, 121(2):219–245, 1997.
2. E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.
3. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, pages 266–277, Los Altos, CA 94022, USA, Sept. 1999. Morgan Kaufmann Publishers.
4. M. M. Al-Jarrah and I. S. Torsun. An empirical analysis of COBOL programs. *Software–Practice and Experience*, 9:341–359, 1979.
5. J. R. Anderson. Interference: The relationship between response latency and response accuracy. *Journal of Experimental Psychology: Human Learning and Memory*, 7(5):326–343, 1981.
6. J. R. Anderson. *Learning and Memory*. John Wiley & Sons, Inc, second edition, 2000.
7. N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON'98*, pages 213–222, 1998.
8. J. Backus. The history of FORTRAN I, II, and III. *SIGPLAN Notices*, 13(8):165–180, 1978.
9. A. D. Baddeley. How does acoustic similarity influence short-term memory? *Quarterly Journal of Experimental Psychology*, 20:249–264, 1968.
10. A. D. Baddeley. *Essentials of Human Memory*. Psychology Press, 1999.
11. H. P. Bahrick. Semantic memory content in permastore: Fifty years of memory for Spanish learned in school. *Journal of Experimental Psychology: General*, 113(1):1–26, 1984.
12. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.
13. M. C. Baker. *The Atoms of Language*. Basic Books, 2001.
14. T. Ball and J. R. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, June 1993.
15. V. A. Bell and P. N. Johnson-Laird. A model theory of modal reasoning. *Cognitive Science*, 22(1):25–51, 1998.
16. B. Berlin and P. Kay. *Basic Color Terms*. Berkeley: University of California Press, 1969.
17. R. Bhargava, J. Rubio, S. Kannan, and L. K. John. Understanding the impact of X86/NT computing on microarchitecture. In L. K. John and A. M. G. Maynard, editors, *Characterization of Contemporary Workloads*, chapter 10, pages 203–228. Kluwer Academic Publishers, 2001.
18. J. M. Bieman and V. Murdock. Finding code on the world wide web: A preliminary investigation. In *Proceedings First International Workshop on Source Code Analysis and Manipulation (SCAM2001)*, pages 73–78, 2001.
19. R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.
20. G. H. Bower, M. C. Clark, A. M. Lesgold, and D. Winzenz. Hierarchical retrieval schemes in recall of categorized word lists. *Journal of Verbal Learning and Verbal Behavior*, 8:323–343, 1969.
21. J. D. Bransford and J. J. Franks. The abstraction of linguistic ideas. *Cognitive Psychology*, 2:331–350, 1971.
22. J. D. Bransford and M. K. Johnson. Contextual prerequisites for understanding: Some investigations of comprehension and recall. *Journal of Verbal Learning and Verbal Behavior*, 11:717–726, 1972.
23. H. D. Brown. Categories of spelling difficulty in speakers of English as a first and second language. *Journal of Verbal Learning and Verbal Behavior*, 9:232–236, 1970.
24. M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 222–233. ACM Press, June 2002.
25. B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.
26. B. Calder, D. Grunwald, and A. Srivastava. The predictability of branches in libraries. Technical Report Research Report 95/6, Western Research Laboratory - Compaq, 1995.
27. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
28. J. I. D. Campbell. On the relation between skilled performance of simple division and multiplication. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23(5):1140–1159, 1997.
29. J. I. D. Campbell and D. J. Graham. Mental multiplication skill: Structure, process, and acquisition. *Canadian Journal of Psychology*, 39(2):338–366, 1985.
30. A. K. Carter and C. G. Clopper. Prosodic and morphological effects on word reduction in adults: A first report. Technical Report Progress Report No. 24 (2000), Speech Research Laboratory, Indiana University, 2000.
31. E. Caspi. Empirical study of opportunities for bit-level specialization in word-based programs. Thesis (m.s.), University of California, Berkeley, 2000.
32. K. A. Cassell. Tools for the analysis of large PROLOG programs. Thesis (m.s.), University of Texas at Austin, Austin, TX, 1985.
33. J. P. Cavanagh. Relation between the immediate memory span and the memory search rate. *Psychological Review*, 79(6):525–530, 1972.
34. M. Celce-Murcia and D. Larsen-Freeman. *The Grammar Book: An ESL/EFL Teacher's Course*. Heinle & Heinle, second edition, 1999.
35. S. M. Chambers and K. I. Forster. Evidence for lexical access in a simultaneous matching task. *Memory & Cognition*, 3(5):549–559, 1975.
36. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software–Practice and Experience*, 22(5):349–369, 1992.

37. B.-C. Cheng. *Compile-Time Memory Disambiguation for C Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

38. R. J. Chevance and T. Heidet. Static profile and dynamic behavior of COBOL programs. *SIGPLAN Notices*, 13(4):44–57, Apr. 1978.

39. D. Citron, D. Feitelson, and L. Rudolph. Accelerating multimedia processing by implementing memoing in multiplication and division units. In *Proceedings of 8^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 252–261, 1998.

40. D. Citron and D. G. Feitelson. Hardware memoization of mathematical and trigonometric functions. Technical Report 2000-5, Hebrew University of Jerusalem, Mar. 2000.

41. D. W. Clark and C. C. Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–87, Feb. 1977.

42. G. Cohen. Why is it difficult to put names to faces? *British Journal of Psychology*, 81:287–297, 1990.

43. A. M. Collins and M. R. Quillian. Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 8:240–247, 1969.

44. B. Comrie. *Language Universals and Linguistic Typology*. Blackwell, second edition, 1989.

45. D. A. Connors, Y. Yamada, and W. mei W. Hwu. A software-oriented floating-point format for automotive control systems. In *Workshop on Compiler and Architecture Support for Embedded Computing Systems (CASES'98)*, 1998.

46. R. Conrad. Order error in immediate recall of sequences. *Journal of Verbal Learning and Verbal Behavior*, 4:161–169, 1965.

47. V. J. Cook. *Inside Language*. Arnold, 1997.

48. N. S. Coulter and N. H. Kelly. Computer instruction set usage by programmers: An empirical investigation. *Communications of the ACM*, 29(7):643–647, July 1986.

49. M. Daneman and P. A. Carpenter. Individual differences in working memory and reading. *Journal of Verbal Learning and Verbal Behavior*, 19:450–466, 1980.

50. M. Daneman and P. A. Carpenter. Individual differences in integrating information between and within sentences. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 9(4):561–584, 1983.

51. J. W. Davidson, J. R. Rabung, and D. B. Whalley. Relating static and dynamic machine code measurements. Technical Report CS-89-03, Department of Computer Science, University of Virginia, July 13 1989.

52. C. B. De Soto, M. London, and S. Handel. Social reasoning and spatial paralogic. *Journal of Personality and Social Psychology*, 2(4):513–521, 1965.

53. S. Dehaene and R. Akhavein. Attention, automaticity, and levels of representation in number processing. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21(2):314–326, 1995.

54. D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, Aug. 1993.

55. C. DiMarco, G. Hirst, and M. Stede. The semantic and stylistic differentiation of synonyms and near-synonyms. In *AAAI Spring Symposium on Building Lexicons for Machine Translation*, pages 114–121, Mar. 1993.

56. R. Dirven. Dividing up physical and mental space into conceptual categories by means of English prepositions. In C. Zelinksy-Wibbelt, editor, *Natural Language processing (vol. 3, The Semantics of Prepositions)*, pages 73–97. Mouton de Gruyter, 1993.

57. D. J. Dooling and R. E. Christiaansen. Episodic and semantic aspects of memory for prose. *Journal of Experimental Psychology: Human Learning and Memory*, 3(4):428–436, 1977.

58. W. H. Eichelman. Familiarity effects in the simultaneous matching task. *Journal of Experimental Psychology*, 86(2):275–282, 1970.

59. J. L. Elshoff. A numerical profile of commercial PL/I programs. *Software–Practice and Experience*, 6:505–525, 1976.

60. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.

61. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999.

62. J. Epelboim, J. R. Booth, R. Ashkenazy, A. Taleghani, and R. M. Steinmans. Fillers and spaces in text: The importance of word recognition during reading. *Vision Research*, 37(20):2899–2914, 1997.

63. M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

64. W. K. Estes. *Classification and Cognition*. Oxford University Press, 1994.

65. L. H. Etzkorn, L. L. Bowen, and C. G. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(1):1–18, 1999.

66. J. S. B. T. Evans, J. L. Barston, and P. Pollard. On the conflict between logic and belief in syllogistic reasoning. *Memory & Cognition*, 11(3):295–306, 1983.

67. C. J. Fillmore. Topics in lexical semantics. In R. W. Cole, editor, *Current Issues in Linguistic Theory*, pages 76–138. Indiana University Press, 1977.

68. B. Fluri, M. Würsch, and H. C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE)*, page ???, Oct. 2007.

69. J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 192–203, 1999.

70. M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25^{th} Annual International Symposium on Microarchitecture (MICRO-25)*, pages 236–245, 1992.

71. R. Frost, L. Katz, and S. Bentin. Strategies for visual word recognition and orthographical depth: A multilingual comparison. *Journal of Experimental Psychology: Human Perception and Performance*, 13(1):104–115, 1987.

72. W. Gellerich, M. Kosiol, and E. Ploedereder. Where does GOTO go to? In *Reliable Software Technology —Ada-Europe 1996*, volume 1088 of *LNCS*, pages 385–395. Springer, 1996.

73. A. C. Graesser, S. B. Woll, D. J. Kowalski, and D. A. Smith. Memory for typical and atypical actions in scripted activities. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):503–515, 1980.

74. J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In L. Wills, I. Baxter, and E. Chikofsky, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 227–238. IEEE Computer Society Press, Nov. 1996.

75. D. Green and P. Meara. The effects of script on visual search. *Second Language Research*, 3(2):102–118, 1987.

76. K. R. Hammond, R. M. Hamm, J. Grassia, and T. Pearson. Direct comparison of the efficacy of intuitive and analytical cognition in expert judgment. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17:753–770, Sept. 1987.

77. W. S. Harley. *Associative Memory in Mental Arithmetic*. PhD thesis, Johns Hopkins University, Oct. 1991.

78. M. W. Harm. *Division of Labor in a Computational Model of Visual Word Recognition*. PhD thesis, University of Southern California, Aug. 1998.

79. M. J. Harrold, J. A. Jones, and G. Rothermel. Empirical studies of control dependence graph size for C. *Empirical Software Engineering Journal*, 3(2):203–211, Mar. 1998.

80. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.

81. T. P. Hill. A statistical derivation of the significant-digit law. *Statistical Science*, 10:354–363, 1996.

82. T. P. Hill. The first-digit phenomenon. *American Scientist*, 86:358–363, July-Aug. 1998.

83. R. M. Hogarth and H. J. Einhorn. Order effects in belief updating: The belief-adjustment model. *Cognitive Psychology*, 24:1–55, 1992.

84. L. M. Horowitz. Free recall and ordering of trigrams. *Journal of Experimental Psychology*, 62(1):51–57, 1961.

85. M. W. Howard and M. J. Kahana. When does semantic similarity help episodic retrieval? *Journal of Memory and Language*, 46:85–98, 2002.

86. W. W. Hsu, A. J. Smith, and H. C. Young. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. Technical Report UCB/CSD-99-1071, University of California, Berkeley, Nov. 1999.

87. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.

88. P. J. Jalics. COBOL on a PC: A new perspective on a language and its performance. *Communications of the ACM*, 30(2):142–154, Feb. 1987.

89. R. Jordan, R. Lotufo, and D. Argiro. *Khoros Pro 2001 Student Version*. Khoral Research, Inc, 2000.

90. D. Kahneman and A. Tversky. On the psychology of prediction. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 4, pages 48–68. Cambridge University Press, 1982.

91. D. Kahneman and A. Tversky. Subjective probability: A judgment of representativeness. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 3, pages 32–47. Cambridge University Press, 1982.

92. D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 2, pages 17–43. Cambridge University Press, 1999.

93. T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

94. B. Kelk. Letter frequency rankings for various languages. www.bckelk.uklinux.net/words/etaoin.html, 2003.

95. W. Kintsch and J. Keenan. Reading rate and retention as a function of the number of propositions in the base structure of sentences. *Cognitive Psychology*, 5:257–274, 1973.

96. D. Klahr, W. G. Chase, and E. A. Lovelace. Structure and process in alphabetic retrieval. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 9(3):462–477, 1983.

97. J. Klayman and Y.-W. Ha. Confirmation, disconfirmation, and information in hypothesis testing. *Psychological Review*, 94(2):211–228, 1987.

98. J. L. Knetsch. The endowment effect and evidence of nonreversible indifference curves. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 9, pages 171–179. Cambridge University Press, 1999.

99. D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1:105–133, 1971.

100. P. A. Kolers. Reading A year later. *Journal of Experimental Psychology: Human Learning and Memory*, 2(3):554–565, 1976.

101. N. kuan Tsao. On the distribution of significant digits and roundoff errors. *Communications of the ACM*, 17(5):269–271, May 1974.

102. W. Labov. The boundaries of words and their meaning. In C.-J. N. Bailey and R. W. Shuy, editors, *New ways of analyzing variation of English*, pages 340–373. Georgetown Press, 1973.

103. K. Laitinen. *Natural naming in software development and maintenance*. PhD thesis, University of Oulu, Finland, Oct. 1995. VTT Publications 243.

104. K. Laitinen, J. Taramaa, M. Heikkilä, and N. C. Rowe. Enhancing maintainability of source programs through disabbreviation. *Journal of Systems and Software*, 37:117–128, 1997.

105. B. L. Lambert, K.-Y. Chang, and P. Gupta. Effects of frequency and similarity neighborhoods on phamacists' visual perception of drug names. *Social Science and Medicine*, 57(10):1939–1955, Nov. 2003.

106. D. Lee, J.-L. Baer, B. Bershad, and T. Anderson. Reducing startup latency in web and desktop applications. Technical Report TR-99-03-01, University of Washington, Department of Computer Science and Engineering, Mar. 1999.

107. G. Leech, R. Garside, and M. Bryant. CLAWS4: The tagging of the British national corpus. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING 94)*, pages 622–628, Apr. 1994.

108. G. Leech, P. Rayson, and A. Wilson. *Word Frequencies in Written and Spoken English*. Pearson Education, 2001.

109. G. E. Legge, T. S. Klitz, and B. S. Tjan. Mr. Chips: An ideal-observer model of reading. *Psychological Review*, 104(3):524–553, 1997.

110. K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50:1174–1190, 2001.

111. S. Lichtenstein and B. Fishhoff. Do those who know more also know more about how much they know? *Organizational Behavior and Human Performance*, 20:159–183, 1977.

112. D. Lin. Automatic retrieval and clustering of similar words. In *Proceedings of Coling/ACL-98*, pages 768–774, 1998.

113. M. A. Linton and R. W. Quong. A macroscopic profile of program compilation and linking. *IEEE Transactions on Software Engineering*, 15(4):427–436, Apr. 1989.

114. M. Magnus. *What's in a Word? Studies in Phonosemantics*. PhD thesis, Norwegian University of Department of Linguistics Science and Technology, Apr. 2001.

115. J. M. Mandler and N. S. Johnson. Remembrance of things parsed: Story structure and recall. *Cognitive Psychology*, 9:111–151, 1977.

116. A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the* 16^{th} *IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.

117. A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPLOS-VI: Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Oct. 1994.

118. M. McCloskey, A. Washburn, and L. Felch. Intuitive physics: The straight-down belief and its origin. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 9(4):636–649, 1983.

119. D. McFadden. Rationality for economists? *Journal of Risk and Uncertainty*, 19:73–105, 1999.

120. T. P. McNamara, J. K. Hardy, and S. C. Hirtle. Subjective hierarchies in spatial memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(2):211–227, 1989.

121. Microchip. *PIC18CXX2 High-Performance Microcontrollers with 10-Bit A/D*, ds39026b edition, 1999.

122. G. A. Miller, J. S. Bruner, and L. Postman. Familiarity of letter sequences and tachistoscope identification. *The Journal of General Psychology*, 50:129–139, 1954.

123. G. A. Miller and S. Isard. Free recall of self-embedded English sentences. *Information and Control*, 7:292–303, 1964.

124. M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analysis and potential applications in program understanding and optimization. Technical Report UW CSE Technical Report 01-03-01, University of Washington, Mar. 2001.

125. A. F. Monk and C. Hulme. Errors in proofreading: Evidence for the use of word shape in word recognition. *Memory & Cognition*, 11(1):16–23, 1983.

126. G. L. Murphy and D. L. Medin. The role of theories in conceptual coherence. *Psychological Review*, 92(3):289–315, 1985.

127. P. Muter and E. E. Johns. Learning logographies and alphabetic codes. *Applied Cognitive Psychology*, 4:105–125, 1985.

128. I. Neamtiu. Detailed break-down of general data provided in paper[129] kindly supplied by first author. Jan. 2008.

129. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.

130. R. E. Nisbett, D. H. Krantz, C. Jepson, and Z. Kunda. The use of statistical heuristics in everyday inductive reasoning. *Psychological Review*, 90(4):339–363, 1983.

131. S. F. Oberman. Design issues in high performance floating point arithmetic units. Technical Report CSL-TR-96-711, Stanford University, Dec. 1996.

132. K. R. Paap, S. L. Newsome, and R. W. Noel. Word shape's in poor shape for the race to the lexicon. *Journal of Experimental Psychology: Human Perception and Performance*, 10(3):413–428, 1984.

133. S. E. Palmer. *Vision Science: Photons to Phenomenology*. The MIT Press, 1999.

134. B. K. Pasquale and G. C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–397, Nov. 1993.

135. J. W. Payne, J. R. Bettman, and E. J. Bettman. *The Adaptive Decision Maker*. Cambridge University Press, 1993.

136. M. J. Pazzani. Influence of prior knowledge on concept acquisition: Experimental and computational results. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 17(3):416–432, 1991.

137. R. Peereman and A. Content. Orthographic and phonological neighborhood in naming: Not all neighbors are equally influential in orthographic space. *Journal of Memory and Language*, 37:382–410, 1997.

138. D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 1993 European Software Engineering Conference*, pages 48–67, 1993.

139. J. J. Pollock and A. Zamora. Collection and characterization of spelling errors in scientific and scholarly text. *Journal of the American Society for Information Science*, 34(1):51–58, 1983.

140. P. T. Quinlan and R. N. Wilton. Grouping by proximity or similarity? Competition between gestalt principles in vision. *Perception*, 27:417–430, 1998.

141. A. Ramírez, J.-L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. In *IEEE International Conference on Parallel Processing (ICPP99)*, 1999.

142. E. J. Ratliff. Decreasing process memory requirements by overlapping run-time stack data. Thesis (m.s.), Florida State University, College of Arts and Sciences, 1997.

143. J. Reason. *Human Error*. Cambridge University Press, 1990.

144. RedHill Consulting, Pty. Simian - similarity analyser, v 2.0.2. www.redhillconsulting.com.au, 2004.

145. E. D. Reichle, A. Pollatsek, D. L. Fisher, and K. Rayner. Towards a model of eye movement control in reading. *Psychological Review*, 105(1):125–157, 1998.

146. C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *USENIX Winter 1993 Technical Conference Proceedings*, pages 405–420, Jan. 1993.

147. D. D. Salvucci. An integrated model of eye movements and visual encoding. *Cognitive Systems Research*, 1(4):201–220, 2001.

148. L. H. Shaffer and J. Hardwick. Typing performance as a function of text. *Quarterly Journal of Experimental Psychology*, 20:360–369, 1968.

149. R. N. Shepard, C. I. Hovland, and H. M. Jenkins. Learning and memorization of classifications. *Psychological Monographs: General and Applied*, 75(15):1–39, 1961.

150. J. A. Sloboda. Visual imagery and individual differences in spelling. In U. Frith, editor, *Cognitive Processes in Spelling*, chapter 11, pages 231–248. Academic Press, 1980.

151. S. Sloman and D. A. Lagnado. Counterfactual undoing in deterministic causal reasoning. In *Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society*, 2002.

152. R. L. Solso and J. F. King. Frequency and versatility of letters in the English language. *Behavior Research Methods & Instrumentation*, 8(3):283–286, 1976.

153. F. Spadini, M. Fertig, and S. J. Patel. Characterization of repeating dynamic code fragments. Technical Report CRHC-02-09, University of Illinois at Urbana-Champaign, 2002.

154. S. Srivastava, M. Hicks, J. S. Foster, and P. Jenkins. Modular information hiding and type-safe linking for C. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 3–14, Apr. 2007.

155. K. E. Stanovich. *Who Is Rational?* Lawrence Erlbaum Associates, 1999.

156. M. W. Stephenson. Bitwise: Optimizing bitwidths using data-range propagation. Thesis (m.s.), M.I.T, Cambridge, MA, USA, May 2000.

157. S. Sternberg. Memory-scanning: Mental processes revealed by reaction-time experiments. *American Scientist*, 57(4):421–457, 1969.

158. A. Stevens and P. Coupe. Distortions in judged spatial relations. *Cognitive Psychology*, 10:422–437, 1978.

159. M. Stiff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. Technical Report Technical Report BL0113590-990202-03, Bell Laboratories, 1999.

160. L. A. Streeter, J. M. Ackroff, and G. A. Taylor. On abbreviating command names. *The Bell System Technical Journal*, 62(6):1807–1826, 1983.

161. D. W. Sweeney. An analysis of floating-point addition. *IBM Systems Journal*, 4(1):31–42, 1965.

162. P. F. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, pages 324–332, June 1998.

163. J. Thiyagalingam. *Alternative Array Storage Layout for Regular Scientific Programs*. PhD thesis, Imperial College, University of London, June 2005.

164. J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays, yet? *Concurrency and Computation: Practice & Experience*, 18(11):1509–1539, Sept. 2006.

165. P. Thompson. Margaret Thatcher: a new illusion. *Perception*, 9:483–484, 1980.

166. A. Treisman and J. Souther. Search asymmetry: A diagnostic for preattentive processing of separable features. *Journal of Experimental Psychology: General*, 114(3):285–310, 1985.

167. L. M. Trick and Z. W. Pylyshyn. What enumeration studies can show us about spatial attention: Evidence for limited capacity preattentive processing. *Journal of Experimental Psychology: Human Perception and Performance*, 19(2):331–351, 1993.

168. J. Turley. Embedded processors. www.extremetech.com, Jan. 2002.

169. A. Tversky and I. Simonson. Context-dependent preferences. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 29, pages 518–527. Cambridge University Press, 1999.

170. G.-R. Uh and D. B. Whalley. Effectively exploiting indirect jumps. *Software–Practice and Experience*, 29(12):1061–1101, Oct. 1999.

171. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.

172. C. Ware. *Information Visualization Perception for Design*. Morgan Kaufmann Publishers, 2000.

173. R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 45–54, Sept. 2003.

174. J. Yang and R. Gupta. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems*, 2(3):1–27, 2002.

175. J. J. Yi and D. J. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *International Conference on Computer Design (ICCD'02)*, pages 462–467, Sept. 2002.