Culture and formal education issues

Discussion and proposed guidelines

Derek M. Jones
derek@knosof.co.uk

1 Introduction

Most of the knowledge and experience that exists in the heads of software developers was obtained while performing non-software development related tasks. For instance, talking to other people, reading prose (i.e., not source code), and solving mathematical problems as a student. The significantly greater quantity of this cultural knowledge and formal education experience, in comparison to software development task experience, means it will be the largest significant factor in developer performance.

The source code making up a program may be written and maintained by developers coming from a range of cultures and having a range of educational backgrounds. To what extent do differences in the cultural and educational background of developers impact fault rates and is it possible to frame guideline recommendations that address this root cause of faults?

The following subsections highlight two computer language usage areas where cultural and educational factors are likely to effect developer performance: the semantic information associated with identifier character sequences and interpreting unparenthesized expressions (a recent study[5] found a 35% error rate).

The material in this proposal has been distilled from that appearing in sentence 0, 787, and 933 of "The New C Standard: An economic and cultural commentary" by Derek M. Jones (available via http://www.knosof.co.uk/cbook/cbook.html).

2 A well chosen identifier

A well chosen identifier character sequence is easy to remember, difficult to confuse with other character sequences, and triggers semantic associations in a developers head which help reduce the effort needed to comprehend the source code containing it.

Studies have found that the amount of information people can remember about a character sequence is relatively constant. However, a persons long exposure to a particular language (e.g., English) teaches them a great deal about the common letter sequences that occur in its written form. Experienced readers use the character sequence predictability to process words from their native language much more efficiently than words from languages having different characteristics. The rate of information extraction does not increase, readers simply learn to make use of the redundancy contained within it. The entropy of English has been estimated as 1.75 bits per character[11] (a random selection from 26 letters or space would have an entropy of 4.75 bits per character).

Table .1: 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[7]).

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

One study[7] measured the amount of information subjects remembered about a briefly presented nonword. The nonwords were constructed so as to have different orders of approximation to existing English words (see Table .1). Subjects saw a single nonword for durations ranging from 10 to 500 ms and then had to write down the letters seen and their position in the nonword.

The results (see Figure .1) show a consistent difference among the order of approximation for all presentation times. The performance difference occurred because the higher-order letter sequences had a lower

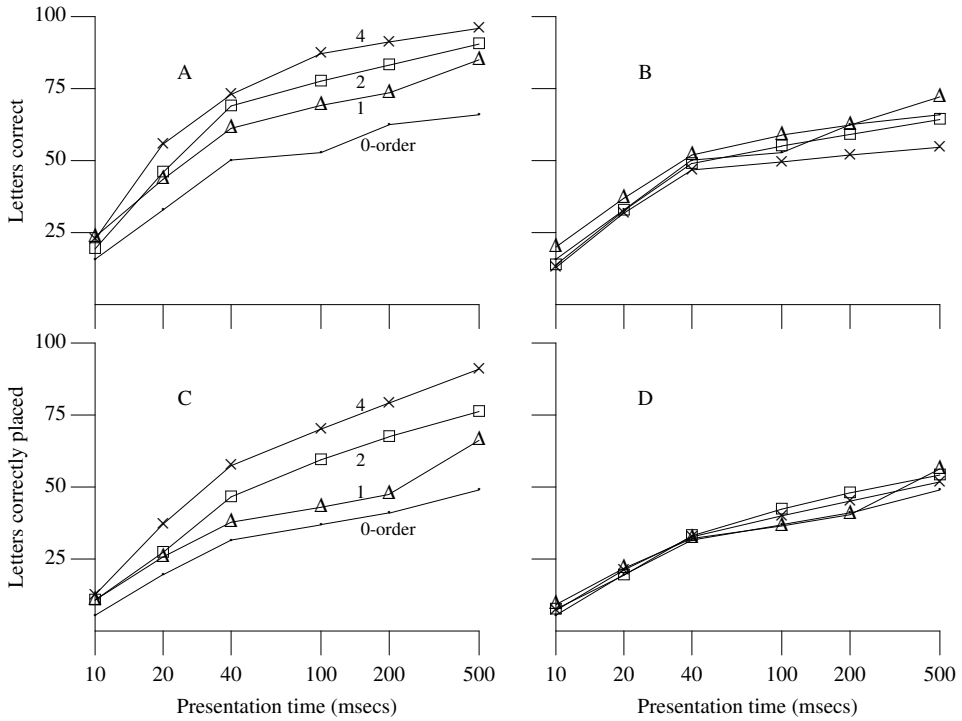

Figure .1: Number of correct letters regardless of position (A), and number of correct letters placed in the correct position (C). Normalizing for information content, the corresponding results are (B) and (D), respectively. Plotted lines denote 0-, 1-, 2-, and 4-order approximations to English words (see Table .1). Adapted from Miller, Bruner, and Postman.[7]

information content for the English language speaking subjects. Had the subjects not been native English speakers, but Japanese speakers for instance, they would have had no knowledge of English letter frequency and the higher-order letter sequences would have contained just as much information as the lower-order ones.

The issue of confusion between different character sequences is covered in another proposal.[5]

A character sequence triggers semantic associations in a developers head when it is recognised as a word, combination of words, or abbreviation that has been previously remembered. Extensive experience using the words from a native language provides a developer with a huge repotoire of word knowledge.[.1]

More targeted semantic information can be specified by using character sequences formed by concatenating two or more known words, abbreviations, or acronyms. The interpretation given to a sequence of these conceptual units, by a developer, may depend on their relative ordering. For instance, `widget_num` or `num_widget` might be considered, by native English speakers, to denote a number assigned to a particular widget and some count of widgets, respectively. Such an interpretation is dependent on knowledge of English word order and conventions for abbreviating sentences (e.g., "widget number 27" and "number of widgets"). This distinction is subtle and relies on a very fine a point of interpretation which could quite easily be given the opposite interpretation.

2.1 Faults cause by different languages

During the lifetime of a program its source code may be worked on by developers having different first languages (their native, or mother tongue). While many developers communicate using English, it is not

[.1] English contains (per *Webster's Third International Dictionary*, the largest dictionary not based on historical principles) around 54,000 word families (*excited*, *excites*, *exciting*, and *excitement* are all part of the word family having the headword *excite*). A variety of studies[8] have shown that a few thousand word families account for more than 90% of words encountered in newspapers and popular books.

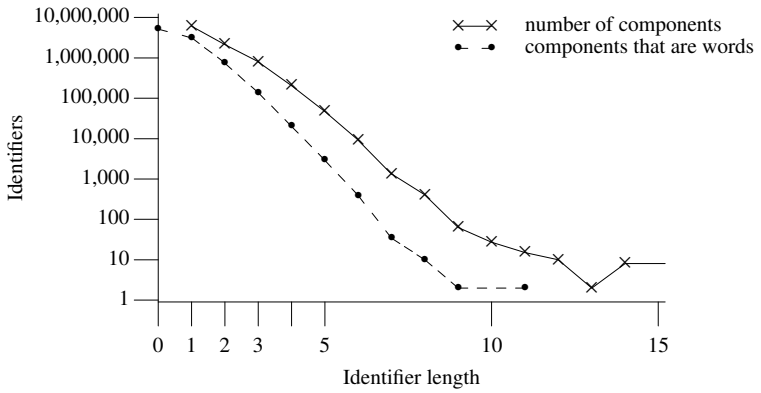

Figure .2: Identifiers containing a given number of *components* (where a component is defined as a character sequence delimited by one or more underscore characters, _, the start of the identifier, or its ending, e.g., the identifier `big_blk_proboscis` is considered to contain three components, one of which is a word). 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.

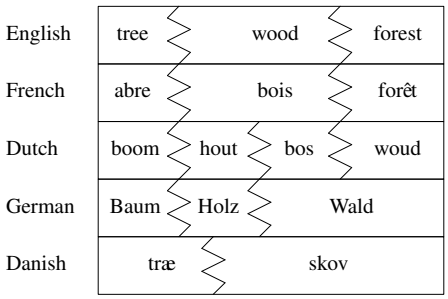

Figure .3: The relationship between words for tracts of trees in various languages. The interpretation given to words (boundary indicated by the zigzags) in one language may overlap that given in other languages. Adapted from DiMarco, Hirst, and Stede.[4]

always their first language. There are native speakers of many different human languages writing source code. The availability of cheaper labour outside of the industrialized nations is slowly shifting developers' native language away from those nations' languages to Mandarin Chinese, Hindi/Urdu, and Russian.

If English was good enough for Jesus, it is good enough for me. (attributed to various U.S. politicians)

No empirical data on the number of mistakes made by developers, whose root cause comes from differences in native language between the author and reader of source code, is available and is unlikely to be available in the near future.

Developers may read one character sequence as-if it were another, because of expectations derived from experience reading character sequences containing many common regularities. The issue of confusion between character sequences is covered in another proposal,[5] and this is another root cause of this kind of mistake.

If the authors of a program encoded semantic information into the character sequences used for identifiers, it is possible that subsequent readers will misinterpret the intended meaning. An example is provided by Figure .3, which shows how words in different languages draw boundaries for possible groupings of trees at different points in a continuum.

Another example of where differences in shared knowledge produce differences in performance is the treatment of time in English and Mandarin Chinese. English predominantly treats time as if it were horizontal (e.g., "ahead of time", "push back the deadline"), while Mandarin often, but not always, treats it as being vertical (an English example of vertical treatment is "passes down the generations").[10] One study[2] found

that English subjects responded more quickly (by approximately 10%) to a question about dates (e.g., "Does March come before April?") if the previous question they had answered had involved a horizontal scenario (e.g., "X is ahead of Y") than if the previous question had involved a vertical scenario (e.g., "X is below Y"). These results were reversed when the subjects were Mandarin speakers and the questions were in Mandarin.

2.2 Native language recommendations

The obvious recommendation is to require that developers not make use of any semantic association that is not universal across human culture. Researchers[12] have proposed a list of universal semantic primitives. Unfortunately there are not very many of them (around 55) and nearly all of them are connected with human interaction (which restricts the number of different kinds of applications that can be written, if only these semantic concepts can be used).

Does the recommendation become more practical if we limit ourselves to only the 12 languages spoken by 100 million or more people, rather than the the 3,000 to 6,000 languages spoken on Earth today (see Table .2)?

Table .2: 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). A sentence can roughly be broken down in to a Subject, an Object and a Verb; all six possible permutations are covered by known languages. 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

Even if we limit ourselves to two related languages, basic differences soon emerge. For instance, while English adjectives always precede the noun they modify, French adjectives referring to color or origin follow the noun (most others precede it):

```
une grande voiture jaune
    (big)   (car)  (yellow)
une vielle femme Italienne
    (old) (woman) (Italian)
```

Adjective order may simply be an example of a harmless difference, where phrases might sound odd but the intended meaning is likely to be deduced. For instance, while the phrase "red big cat" is easily understood it does not sound correct. The reason is that native English speakers use a preferred adjective ordering. That is, the relative position of many of them have a consistent ordering (see Table .3).

Table .3: Probability of an adjective occurring at a particular position relative to other adjectives (based on responses from 30 native English speakers). Adapted from Celce-Murcia.[3]

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

An alternative recommendation is to require that all identifiers in a program be chosen from a predefined set of permissible character sequences (a so called *controlled vocabulary*). After all, why should developers be given the freedom to choose which character sequence should be used to denote an identifier?

Some organizations have decided that this is an acceptable solution to the confusion and increased cost of sharing code and data caused by having developers make their individual choice on which character sequence to use for an identifier. For instance, US law enforcement agencies experienced great difficulty in sharing data because different database developers have used different column names to represent the same kind of data. The column names that are to be used across all law enforcement databases is now being decided on (see the Global Justice XML data dictionary at the Justice Standards Clearinghouse http://it.ojp.gov/jsr/public/).

There is an ISO Standard that deals with naming in a database context: "ISO/IEC 11174 Information technology — Specification and standardization of data elements."

Creating a controlled vocabulary, even for a small application domain, requires a great deal of work. Requiring projects to document their own use of any implied semantics, derived from native language use, of all identifiers appearing in the source code of their programs is a more realistic, and practical, recommendation.

LangSpec .1 Assumptions about the natural language expertise of developers who are expected to work with the source code shall be documented.

Cg .2 If any part of an identifier's character sequence has an implied semantic meaning, in the context of the source code containing it, this meaning shall be documented.

The following are some of the kinds of information that might be documented:

- the meaning that intended to be assigned to a word that has more than one possible meaning (i.e., the word is polysemous),
- rules for joining words together to form compound words,
- how words are abbreviated,
- how metaphors are to be interpreted.

2.3 Distributing the information content of an identifier

The order in which words are spoken and the distribution of sounds within individual words has been shown to follow a pattern that reduces the cognitive effort needed by the listener....

3 Formal education

Developers over learn various skills during the time they spend in formal education. Some of these skills include the following:

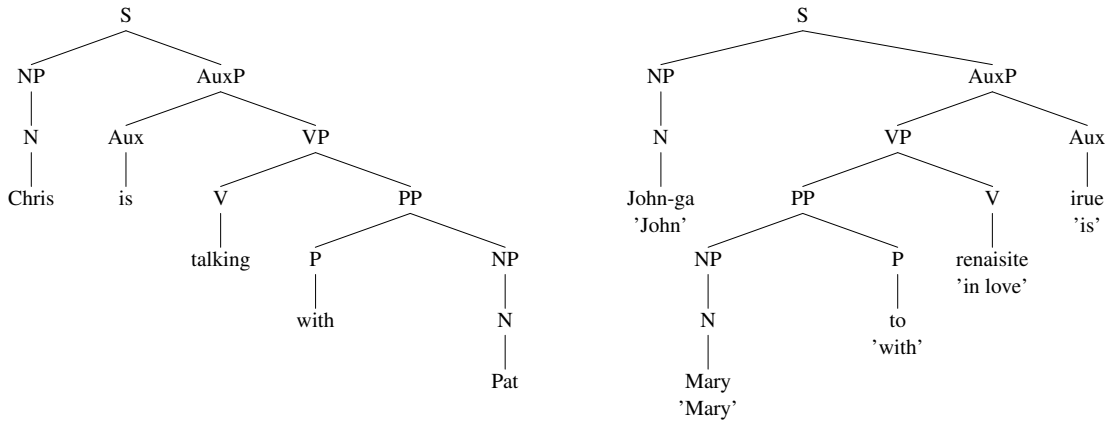

Figure .4: English ("Chris is talking with Pat") and Japanese ("John-ga Mary to renaisite irue") language phrase structure for sentences of similar complexity and structure. While the Japanese structure may seem back-to-front to English speakers, it appears perfectly natural to native speakers of Japanese. Adapted from Baker.[1]

- Reading from left to right. For English it has been shown[9] that readers parse its written form left-to-right, the order in which the words are written. It has not been confirmed that readers of languages written right-to-left parse them in a right-to-left order.
- Reading mathematical expressions. Many science and engineering courses require students to manipulate expressions containing operators that also occur in source code. Students learn, for instance, that in an expression containing a multiplication and addition operator, the multiplication is performed first. Substantial experience is gained over many years in reading and writing such expressions. Knowledge of the ordering relationships between assignment, subtraction, and division also needs to be used on a very frequent basis. Through constant practice, knowledge of the precedence relationships between these operators becomes second nature; developers often claim that they are natural (they are not, it is just constant practice that makes them appear so).

Your author knows of no research studying how developers read expressions (e.g., order of eye movements over an expression). The assumption made by these coding guidelines proposals is that developers' extensive experience reading prose is a significant factor affecting how they read source code. Given the significant differences in the syntactic structure of natural languages (see Figure .4) the possibility of an optimal visual expression organization, which is universal to all software developers, seems remote.

Many students choose to study subjects that fall within one of two broad categories, i.e., science/engineering based or arts/humanities based. People having an arts/humanities education are likely to have substantially less experience in reading and writing mathematical expressions (they might even be unaware of the conventions regarding the precedence of the additive and multiplicative operators).

3.1 Faults relating to parenthesis in expressions

While many developers are likely to have had significant previous experience in reading and writing commonly occurring programming language operators (e.g., binary plus and multiplication), few of developers attain the same level of experience with other operators. Consequently, developer knowledge of the precedence levels of other operators is often faulty (which does not prevent them from being willing to jump to conclusions).

Some languages *solve* the problem of developers forgetting operator precedence levels by giving all operators the same precedence (e.g., APL) and specifying the order of evaluation (right-to-left in the case of APL).

Developer training is not a viable option for solving this problem. While it is possible for people to learn the precedence of all language operators sufficiently well to pass a test, some of this knowledge will degrade

over time through lack of use. Many languages contain operators that are rarely used in practice and it is knowledge of the precedence of these operators that developers are likely to forget the quickest (through of lack of practice).

LangSpec .3 A language specific guideline document shall document its operators and their precedence.

LangSpec .4 A language specific guideline document shall document any differences between the precedence of operators in the language and the precedence commonly used in mathematics.

LangSpec .5 A language specific guideline measure and document shall document the frequency of use of all operators defined by the language.

3.1.1 Associativity

Like precedence, possible developer misunderstandings about how operators associate can be solved using parentheses. Expressions, or parenthesized expressions that consist of a sequence of operators with the same precedence, might be thought to be beyond confusion. As the example below illustrates, how operand type conversion rules or changes in operator associativity might result in behavior that is not expected by a developer.

In the following example, based on the C language, the fact that `j` is added to `i` before `k` is added to the result is not of obvious interest until it is noticed that their types are all different.

```
1  extern float i;
2  extern short j;
3  extern unsigned long k;
4
5  void f(void)
6  {
7  int x, y;
8
9  x = i + j + k;
10
11 y = i / j / k; /* / associates to the left: (i / j) / k */
12
13 i /= j /= k;   /* /= associates to the right: i /= (j /= k) */
14 }
```

Associativity requires that `j` be added to `i`, after being promoted to type **`float`**. The result type of `i+j` (**`float`**) causes `k` to be converted to **`float`** before it is added. The sequence of implicit conversions would have been different had the operators associated differently, or the use of parentheses created a different operand grouping. Dividing `i` by `j`, before dividing the result by `k`, gives a very different answer than dividing `i` by the result of dividing `j` by `k`.

LangSpec .6 A language specific guideline document shall document the associativity of its operators.

LangSpec .7 A language specific guideline document shall document any differences between the associativity of operators in the language and the associativity commonly used in mathematics.

3.1.2 n-ary operators

Some languages support unary operators that can appear on the left or right of the operand they operate on. When all of the unary operators appear on the same side of an operand (e.g., in C, `(char)!-~*++p`) issues of developer unfamiliarity with operator precedence levels and associativity is unlikely to be an issue. When unary operators appear on both sides of an operand some developers may be uncertain about the order in which they are applied (e.g., in C, is `*p++` equivalent to `(*p)++` or `*(p++)`?) In some cases it is even possible for some confusion to be caused when both unary and binary operators appear in the same expression (e.g., `m<-++pq++->m`).

Coding guidelines need to be careful in their use of the term *unary operator*. The meaning, in its mathematical sense (i.e., an operator taking one operand) and as some users of a language may understand it, may be different than the definition that is actually specified by a particular language definition. For instance, the C Standard has a syntax rule called *unary-operator* which does not include the `++` operator, and a subsection called "Unary operators" which does include the `++` operator but not the cast operator.

LangSpec .8 A language specific guideline document shall document all of the operators that it considers to be unary operators.

While some language standards (e.g., C) use the term *binary operator*, they never define it. Instead they rely on the definition contained in other standards (e.g., ISO 2382). Language specifications also differ in whether they treat some tokens as operators or not. For instance, Ada does not treat the token used to denote assignment (e.g., `:=`) as an operator, while C treats the token used to denote assignment (e.g., `=`) as an operator.

LangSpec .9 A language specific guideline document shall document all of the operators that it considers to be binary operators.

3.2 Recommendations

One study[5] found a 35% error rate (random guessing would have produced an error rate of 50%) when experienced developers were asked to parenthesise an expression to show the expected operator precedence. However, no empirical data on the number of faults caused by developers using incorrect operator precedence or associativity during program comprehension is available and is unlikely to be available in the near future.

Parenthesizing all subexpressions of an expression renders a developer's knowledge, right or wrong, of operator precedence and associativity irrelevant. No empirical data on the number of faults caused by developers having to process redundant parenthesis during program comprehension is available and is unlikely to be available in the near future.

Requiring parenthesis around all subexpressions of an expression is unlikely to significantly increase the number of characters appearing in the visible source. Even so these additional (unnecessary) parenthesis do slightly reduce the cognitive resources available to a developer during program comprehension. It is assumed that the reduction in developer cognitive resources does not cause additional faults to be introduced.

Guideline recommendations requiring so called *redundant parenthesis* are often met with howls of protest from developers and more time is probably spent discussing this issue than would be needed to type in the parenthesis.

There are no published figures on faults whose root cause relates to parenthesis (either missing or excessive usage). Tales of faults that might have been averted had parenthesis been used generally involve so called *complex* expressions.

Most expressions are relatively simple[6] with only 2% of expressions containing more than one binary operator.[5] The following proposed guideline recommendations target the less commonly used operators. Because they are uncommon developers are much less likely to be familiar with their relative precedence and requiring the use of parenthesis will not substantially increase the number of visible characters appearing in an expression.

3.2.1 Array indexing, function calls, and member selection

In many languages (e.g., Java, C++, C) function calls, array indexing and member selection are defined to be (binary) operators. While they might technically be operators experience suggests developers do not regard them as such and rarely apply an incorrect relative precedence when they appear in expressions with binary operators.

Developers tend to be less sure of themselves when non-arithmetic unary operators appear with this operations in an expression. For instance, is `&c[i]` equivalent to `(&c)[i]` or `&(c[i])`? In many cases the types of the operand mean that a compiler will only allow one interpretation (the other causing a diagnostic to be generated). However, that does not ensure that the developer and compiler agree on the interpretation chosen.

```
1  x=(c[i]);  // Array indexing is an operator in many languages.
2  y=(f(d));  // () is a function call operator in many languages.
3  z=(a.m);   // Member selection is an operator in many languages.
```

In other languages (e.g., Ada) function calls, array indexing and member selection are not defined as operators, but as part of the syntax for for the most basic kind of reference to storage or value. While they may not be operators their presence in the grammar for expressions means they effectively act like operators having the highest precedence.

If array indexing, function calls, and member selection have, or act as if they have, the highest operator precedence and associate left to right, then developer expectations are met and parenthesis provide no additional information to a reader of the source.

LangSpec .10

If array indexing, function call, and member selection do not have have, or act as if they have, the highest operator precedence and associate left to right, then they will not be considered to be binary operators for the purpose of the parenthesis guideline recommendations.

How should expressions contained with array indexing and function all brackets be treated? The following is the most obvious solution.

For the purpose of these guideline recommendation the expression within

- the square brackets used as an array subscript operator are to be treated as equivalent to a pair of matching parentheses, not as an operator; and
- the arguments in a function invocation are each treated as full expressions and are not considered to be part of the rest of the expression that contains the function invocation for the purposes of the deviations listed.

3.2.2 General issues

The rationale for the following deviations is that developers from a science/engineering background will have over learned the precedence and associativity of some operator combinations. Thus any mistakes they make are unlikely to have confusion over precedence or associativity as their root cause.

The deviations against the following, broadly worded guideline are common combinations of commonly occurring operators combinations.

Cg .11

Each subexpression of a full expression containing more than one binary or ternary operator shall be parenthesized.

Dev .11

A full expression that only contains one or more additive operators and a single assignment operator need not be parenthesized.

Dev .11 A full expression that only contains zero or more multiplication, division, addition, and subtraction operators and a single assignment operator need not be parenthesized.

Dev .11 A full expression that only contains one or more additive operators and a single relational or equality operator need not be parenthesized.

Dev .11 A full expression that only contains zero or more multiplicative and additive operators and a single relational or equality operator need not be parenthesized.

Experience suggests that even people from an engineering background sometimes associate division from right to left. Division is often written in a vertical, rather than a horizon, direction in hand written and printed mathematics, so people have little experience handling those cases where it is only written horizontally.

Cg .12 If the result of a division or remainder operator is the immediate operand of another multiplicative operator, then the two operators shall be separated by at least one parenthesis in the source.

If an expression consists solely of operations involving the binary plus operator, it might be thought that the only issue that need be considered, when ordering operands, is their values. However, there is a second issue that needs to be considered— their type. If the operand types are different, the final result can depend on the order in which they were written. The following guideline is a special case of one that has been deviated against and so will will need to be followed or specifically deviated against.

Cg .13 If the result of an additive operator is the immediate operand of another additive operator, and the operands have different promoted types, then the two operators shall be separated by at least one parenthesis in the source.

3.2.3 Unary operators

In some languages unary operators may appear on both the left and right of their operand. Unless developers regularly have to disambiguate operands that are operated on by both kinds of operator at the same time, it is likely that mistakes will be made (e.g., in C, is `*p++` equivalent to `(*p)++` or `*(p++)`?)

Cg .14 An operand operated on by both prefix and suffix operators shall be parenthesized to indicate the order in which the operations are intended to apply.

In some languages a token may be used to denote both a unary and a binary operator. Unless developers regularly have to disambiguate operands that are operated on by several unary operators, one of which might be a binary operator, it is likely that mistakes will be made (e.g., in C, is `sizeof(int)-1` equivalent to `sizeof((int) -1)` or `(sizeof (int))-1`?)

Cg .15 An operand operated on by a sequence of unary operators, one of which may be a binary operator operators shall be parenthesized to indicate the order in which the operations are intended to apply.

References

Citations added in version 1.0b start at 1449.

1. M. C. Baker. *The Atoms of Language*. Basic Books, 2001.
2. L. Boroditsky. Does language shape thought?: Mandarin and English speaker' conception of time. *Cognitive Psychology*, 43:1–22, 2001.
3. M. Celce-Murcia and D. Larsen-Freeman. *The Grammar Book: An ESL/EFL Teacher's Course*. Heinle & Heinle, second edition, 1999.
4. 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.
5. D. M. Jones. Developer beliefs about binary operator precedence. *C Vu*, 18(4):14–21, Aug. 2006.
6. D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1:105–133, 1971.
7. 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.
8. P. Nation and R. Waring. Vocabulary size, text coverage and word lists. In N. Schmitt and M. McCarthy, editors, *Vocabulary: Description, Acquisition and Pedagogy*, chapter 1.1, pages 6–19. Cambridge University Press, 1998.
9. C. Phillips. *Order and Structure*. PhD thesis, M.I.T., Aug. 1996.
10. A. Scott. The vertical direction and time in Mandarin. *Australian Journal of Linguistics*, 9:295–314, 1989.
11. W. J. Teahan and J. G. Cleary. The entropy of English using PPM-based models. In *IEEE Data Compression Conference*, pages 53–62. IEEE Computer Society Press, 1996.
12. A. Wierzbicka. *Semantics: Primes and Universals*. Oxford University Press, 1996.