Developer beliefs about binary operator precedence

Knowledge Software, Ltd
Farnborough, Hants, England
Derek M. Jones
derek@knosof.co.uk

1 Introduction

A common developer response to a fault being pointed out in their code is to say “What I intended to write was ...”. A knowledge based mistake occurs when a developer acts on a belief they have that does not reflect reality. The act may be writing code that directly or indirectly makes use of this incorrect belief. An example may occur in the expression `x & y == z`, if its author, or a subsequent reader, incorrectly expects it to behave as if it had been written in the form `( x & y ) == z`.

This is the first of a two part article that reports on an experiment carried out during the 2006 ACCU conference investigating both knowledge of binary operator precedence and the consequences of a limited capacity short term memory on subjects performance in recalling information about assignment statements.

It is hoped that this study will provide information that can be used to build a model of how developers judge operator precedence and the impact that different kinds of identifier character sequences have on the cognitive resources needed during program comprehension. Such a model will be of use in analysing source code for possible, knowledge based, coding errors.

This article is split into two parts, the first (this one) provides general background on the study and discusses the results of the relative precedence selection problem, while part two discusses the assignment problem.

2 The hypothesis

Knowledge based mistakes might be broadly divided into those that don’t seem to follow any pattern and those that do (or at least seem to). Here we are interested in finding patterns in the mistakes made by developers in choosing which of two binary operators has the higher precedence.

Studies have consistently found a significant correlation between a person’s performance on a task and the amount of practice they have had performing that task.[2] Based on these findings it is to be expected that there will be a significant correlation between a developer’s knowledge of relative binary operator precedence and the amount of experience they have had handling the respective binary operator pair.

A reader of the visible source only has to make a decision about the relative precedence of binary operators when an operand could have two possible binary operators applied to it. This paper uses the term *operator pair* to refer to the two operators involves in this decision. For instance

```
x + (y * z);    // No reader choice to make because parenthesis make it.

x + a[y * z];   // No reader choice to make because of bracketing effect of [ ]

x + y * z << a;  // Two pairs of binary operators
                 // A choice has to be made about which operator y and z bind to.
x + (y * z) | a; // One pair of binary operators
                 // The parenthesized expression could be the operand of + or |
```

As Table .1 shows, less than 2% of all expressions contain two or more binary operators.

This analysis assumed that subject performance is not influenced by the order in which the two operators in an operator pair occur.

3 Non-source code experience

There are two main situations where developers are required to comprehend expressions and thus gain experience in deducing precedence. These are formal education and reading/writing source code.

Within formal education expressions are encountered when reading and writing mathematical equations. Many science and engineering courses require students to manipulate equations (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 equations. 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).

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 much less experience handling cases where it is written purely horizontally.

4 Measuring source

The following subsection discusses measurements of binary operator usage in the visible source of a number of large C programs (e.g., gcc, idsoftware, linux, netscape, openafs, openMotif, and postgresql). The visible, rather than the preprocessed, source was used as the basis for measurement because we are interested in what a reader of the source sees and has to make decisions about, and not what the compiler has to analyse.

The contents of preprocessor directives were not included in the measurements.

Your author has not yet made sufficiently extensive measurements of the source code of programs written in any other languages for anything worthwhile to be reported here. Some broad brush measurements of Fortran[7] and PL/1[1] source code have been made by other authors.

4.1 Binary operator usage in expressions

The languages C, C++, C#, Java, Perl, PHP, and some other languages share a large common subset of binary operators having the same relative precedence and associativity. These binary operators, with precedence decreasing from left to right are (operators in the same column have the same precedence):

```
High                                                                 Low
 []    *    +    <<    <    ==    &    ^    |    &&    ||   assignment
 ()    /    -    >>    >    !=                              operators
  .    %               <=
 ->                    >=
```

While, technically they are binary operators array subscript, function call, direct and indirect member selection, and the assignment operators are often not thought of as such by developers. For the rest of this paper the term *binary operator* should be read as excluding these operators unless stated otherwise.

The source code contexts included in these measurements were all the ones in which the C grammar permitted an expression (a *full expression* to use C Standard's terminology) to occur (excluding preprocessor directives).

Table .1: Percentage of expressions containing a given number of binary operators in their visible source code. Note that function call, direct and indirect member selection, and assignment operators are not included here as binary operators (although the arguments of function calls and array subscript expressions are counted as separate expressions).

Binary operators	% occurrence	Binary operators	% occurrence
0	92.82	3	0.62
1	5.28	4	0.14
2	0.86	5	0.13

For learning, or reinforcement of existing knowledge, of binary precedence to occur a binary operator pair needs to be encountered. Measurements of occurrences of binary operators in such pairs, rather than all binary operators, is what our hypothesis suggests subject performance should be correlated with. Table .2 lists both sets of measurements.

Table .2: Occurrences, as a percentage of all such operators, of binary operators in the visible C source code. The second column is based on all occurrences of binary operators in expressions (890,423 operators). The third column is based on occurrences of of binary operator pairs in an expression (e.g., `a + b & c * d` contains the operator pairs **+ &** and **& ***, so **&** is counted twice). There were 309,150 binary operator pairs.

Operator	All occurrences of operator	Occurrences in an operator pair
&&	8.55	22.17
+	12.46	13.41
==	17.62	12.07
||	5.23	11.66
×	4.32	6.34
|	5.19	6.25
-	6.27	6.14
!=	8.73	5.65
<	7.59	4.34
>	3.82	3.60
>=	2.18	2.24
/	1.92	2.07
<=	1.34	1.33
&	9.59	0.70
<<	2.67	0.44
%	0.49	0.25
>>	1.84	0.24
^	0.22	0.08

Table .3 lists the percentage occurrence of each operator pair as a percentage of all operators pairs in all expressions..

Table .3: Occurrences of pairs of binary operators as a percentage of all binary operator pairs (the total number of such operator pairs was 154,575). Table arranged so that operators along the top row have higher precedence. The '-' symbol denotes zero pairs.

	×	/	%	+	-	<<	>>	<	>	<=	>=	==	!=	&	^	|	&&
×	1.222																
/	1.218	.116															
%	.006	.005	.001														
+	6.214	1.125	.104	5.344													
-	1.271	.479	.026	3.345	1.405												
<<	.032	.007	.001	.005	.031	.010											
>>	.037	.004	-	.005	.019	.018	.006										
<	.328	.479	.013	1.001	.822	.026	.009	-									
>	.373	.147	.020	1.188	.774	.013	.026	.003	-								
<=	.068	.032	.001	.349	.137	.003	.004	-	-	-							
>=	.126	.068	.004	.421	.298	.010	.006	-	.001	-	-						
==	.086	.049	.160	.352	.469	.028	.052	-	.001	-	-	.001					
!=	.057	.032	.074	.207	.178	.008	.017	.001	-	.001	-	-	-				
&	.003	.007	.001	.006	.005	.083	.115	.001	.001	-	.001	.001	.004	.144			
^	.018	-	-	-	-	.003	.003	-	.001	-	-	-	-	-	.057		
|	.016	.005	.003	.003	.012	.558	.114	-	-	-	-	-	-	.022	-	5.829	
&&	.093	.043	.045	.272	.222	.006	.012	3.466	2.577	1.692	2.530	13.157	7.959	.516	.001	-	5.392
||	.038	.019	.019	.183	.098	.007	.006	2.487	2.054	.362	1.005	8.924	2.685	.245	-	.001	.021

4.2 Use of parentheses in expressions

If the authors of source code always used parenthesis to specify the intended binding of operands to operators, then neither they or subsequent readers would need to have any knowledge of the operator precedence actually specified by a language; the information they needed would always be visible in the source.

For instance, the expression `a * b + c` could be written as `(a * b) + c`. The first expression is defined to be the operator pair *** +**, while the second is defined to be a parenthesized binary operator **(*) +**. In `(a +`

`b << c) * d` there is no parenthesized binary operator because there are two binary operators within the parenthesis. However, there is an operator pair, i.e., `+ <<`.

Measurements (see Table .4) showed that several operator pairs (see * entries) almost always appear with redundant parenthesis to explicitly specifying the intended precedence.

Table .4: Ratio of occurrences of parenthesized binary operators where one of the operators is enclosed in parenthesis, e.g., `(a * b) - c`, and the other is the corresponding operator pair, e.g., `( * ) -` occurs 0.2 times as often as `* -`. The table is arranged so that operators along the top row have highest precedence and any non-zero occurrences in the upper right quadrant refer to uses where parenthesis have been used to change the default operator precedence, e.g., `* ( - )` occurs 0.8 times as often as `* -`. The total number of these parenthesized operator pairs was 102,822 and the total number of operator pairs was 154,575. When there were no corresponding operators pairs in the visible source the entries are marked with a *.

	(×)	(/)	(%)	(+)	(-)	(<<)	(>>)	(<)	(>)	(<=)	(>=)	(==)	(!=)	(&)	(^)	(|)	(&&)	(||)
×	0.1	0.3	9.5	0.2	0.8	2.3	4.4	0.0	0.0	-	-	0.1	0.1	64.8	0.1	-	-	-
/	0.6	1.2	3.8	0.4	1.6	10.5	4.7	-	-	-	-	-	-	1.6	*	-	-	-
%	5.4	10.5	6.0	4.2	4.2	22.0	*	-	-	-	-	-	-	5.0	*	-	-	-
+	0.2	0.2	1.3	0.0	0.3	337.1	101.7	0.0	0.0	-	0.0	0.0	0.1	104.3	*	13.2	0.0	0.0
-	0.2	0.3	3.1	0.1	0.3	11.5	8.5	0.0	0.0	-	0.0	0.0	0.0	45.4	*	0.5	-	0.0
<<	2.8	10.5	34.0	43.1	20.9	1.5	2.2	0.1	-	-	-	0.1	0.2	10.4	2.8	0.1	-	-
>>	3.4	2.0	*	68.9	22.6	1.3	1.3	-	-	-	-	-	-	8.3	2.2	0.0	-	-
<	0.2	0.2	0.5	0.3	0.4	5.6	5.4	-	-	-	-	-	2.0	96.0	*	-	-	-
>	0.2	0.2	0.2	0.3	0.3	3.9	1.6	-	-	-	-	-	*	45.5	2.0	*	0.0	-
<=	0.3	0.4	1.5	0.2	0.4	4.4	5.0	-	-	-	-	-	1.0	*	-	*	-	-
>=	0.2	0.1	1.0	0.2	0.4	4.4	2.9	-	-	-	-	-	-	55.0	-	*	-	-
==	0.2	0.6	0.9	0.2	0.2	1.7	2.0	*	6.0	*	*	9.0	*	4689.0	*	*	0.0	-
!=	0.2	0.4	0.5	0.1	0.2	4.2	3.1	8.0	*	-	*	*	*	487.8	*	*	-	-
&	7.2	2.7	0.5	108.8	168.9	15.1	20.9	3.0	1.0	*	2.0	3.0	1.5	0.3	*	55.9	-	-
^	0.5	*	*	*	*	12.8	38.0	*	2.0	-	-	*	*	*	0.1	*	-	-
|	5.1	2.3	1.2	13.0	6.9	6.6	1.7	*	*	-	*	*	*	67.7	*	0.0	-	1.0
&&	0.1	0.1	0.9	0.2	0.2	3.0	2.1	0.3	0.3	0.4	0.3	0.4	0.5	6.1	18.0	*	0.0	26.0
||	0.1	0.2	0.7	0.1	0.3	2.7	2.9	0.4	0.4	0.3	0.3	0.6	0.7	4.6	*	27.0	25.5	0.0

When parenthesis are used to specify the intended binding of the operands to operators, readers of the source do not need to make use of their precedence knowledge. Thus there is no learning experience and these occurrences are not included in any count used here.

5 The Experiment

The experiment was run by your author during a 45 minute lunchtime session of the 2006 ACCU conference (`http://www.accu.org`) held in Oxford, UK. Approximately 300 people attended the conference, 18 (6%) of whom took part in the experiment. Subjects were given a brief introduction to the experiment, during which they filled out brief background information about themselves, and they then spent 20 minutes working through the problems. All subjects volunteered their time and were anonymous.

One person wrote on the answer sheet that they did not understand the parenthesis problem. This person did not answer any of the parenthesis problems (a few seemed to have been answered, but not in a way that allowed the intent to be unambiguously deduced). Results from 17 subjects were used.

5.1 The problem to be solved

The problem to be solved followed the same format as an experiment performed at a previous ACCU conference and the details can be found elsewhere.[6]

The following is an excerpt of the text instructions given to subjects:

The task consists of remembering the value of three different variables and recalling these values later. The variables and their corresponding values appear on one side of the sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. Read the variables and the values assigned to them as you might when carefully reading lines of code in a function definition.
2. Turn the sheet of paper over. Please do **NOT** look at the assignment statements you have just read again, i.e., once a page has been turned it stays turned.
3. At the top of the page there is a list of five expressions. Each expression contains three different operands and two different operators. Insert parenthesis to denote how you think the operators will bind to the operands (i.e., you are inserting redundant parenthesis).
4. You are now asked to recall the value of the variables read on the previous page. There is an additional variable listed that did not appear in the original list.
   - if you remember the value of a variable write the value down next to the corresponding variable,
   - if you feel that, in a real life code comprehension situation, you would reread the original assignment, tick the "would refer back" column of the corresponding variable,
   - if you don't recall having seen the variable in the list appearing on the previous page, tick the "not seen" column of the corresponding variable.

The following is an example of one of the problems seen by subjects. One side of a sheet of paper contained three assignment statements while the second side of the same sheet contained the five expressions and a table to hold the recalled information. A series of X's were written on the second side to ensure that subjects could not see through to identifiers and values appearing on the other side of the sheet. Each subject received a stapled set of sheets containing the instructions and 40 problems (one per sheet of paper).

```
--------------------- first side of sheet starts here ----------------------
 p =  13;
 q =  72;
 r =  29;
--------------------- second side of sheet starts here ---------------------
x  +  y  *  z
x  /  y  ==  z
x  !=  y  -  z
x  %   y  ^  z
x  <<  y  -  z
                 remember    would refer back     not seen
 q       =      ____            __                  __
 p       =      ____            __                  __
 s       =      ____            __                  __
 r       =      ____            __                  __
```

End of excerpt of text instructions given to subjects.
The parenthesis insertion task can be viewed as either a time filler for the assignment remember/recall problem, or as the main subject of the experiment. IN the latter case the purpose of the assignment problem is to make it difficult for subjects to track answers they had given to previous, related operator pair, problems. Subject's performance on the parenthesis task is discussed in this article (the first of two).

5.2 The parenthesis problems

Based on the results of the 2004 ACCU experiment[6] it was anticipated that on average subjects would answer around 20 questions. By practicing on himself the author concluded that answering five parenthesis problems would take approximately the same amount of time as answering the `if-statement` problem used in the 2004 experiment. This gave an estimated 100 answers per subject (subjects actually averaged 123.5 answers).

To simplify the analysis the it was decided subjects would encounter every operator pair in a problem before they say the same pair again. If subjects were expected to answer around 100 problems then some operators needed to be removed from consideration. The binary operators used had 10 levels of precedence (some shared the same precedence levels), giving 90 possible combinations of pairs of operators if we require the operator pairs to have different relative precedence. It was decided to include a small number of problems where the operator pairs had the same precedence and the same precedence pairs `+`/`-` and `==`/`!=` were chosen. The operators `*`, `/`, and `%` share the same precedence and span the range of frequency of occurrence in source code (see Table .2) and were all included for this reason.

The `<` operator was chosen to represent the relational operators because it is the most commonly occurring of the four (just over 50% of all relational operators), and the `<<` operator because it is slightly more common than `>>`.

If one set of two operators having the same precedence could appear there are 110 possible pairs, for two sets of two operators having the same precedence there are 121 possible pairs, and so on. The problems were automatically generated using an `awk` script which was parameterized by a data file that specified when to use pairs of operators having the same precedence.

All possible operator pairs (from which ever set of operators was selected by the data file) were generated and their order randomised, as was the ordering of the individual operators of a pair. The generation process was then repeated until enough precedence problems had been generated for 40 subjects (8,000 problems). Two hundred problems were successively taken from this list for each subject.

The binary operators that appeared in the problems seen by subjects, with precedence decreasing from left to right are (operators in the same column have the same precedence):

```
*    +    <<    <    ==    &    ^    |    &&    ||
/    -               !=
%
```

The number of problems generated for the `%` operator was significantly less than for the other operators and the `!=`, `-`, and `/` operators did did not occur as frequently as the other operators (see Table .5). This difference was driven by the data script used to control the generation of the problems. At the time it was thought to be a good idea.

6 Results

6.1 Subject experience

Traditionally, developer experience is measured in number of years of employment. In practice the quantity of source code (lines is one such measure) read and written by a developer (interaction with source code overwhelmingly occurs in its written, rather than spoken, form) is likely to be a more accurate measure of source code experience than time spent in employment. Developer interaction with source code is rarely a social activity (a social situation occurs during code reviews) and the time spent on these social activities may be small enough to ignore. The problem with this measure is that it is very difficult to obtain reliable estimates of the amount of source read and written by developers. This issue was also addressed in studies performed at a previous ACCU conference.[5, 6]

It has to be accepted that reliable estimates of lines read/written are not likely to be available until developer behavior is closely monitored (e.g., eye movements and key presses) over an extended period of time.

A plot of problems answered against experience for both the 2004 or 2006 (see Figure .1) experiments did not show any correlation between years of experience and number of problems answered. While a least squares fit of years of experience against percentage of incorrect answers shows a slight upward trend, the Pearson r correlation coefficient is not significant at the 0.05 level.

6.2 Subject motivation

When reading source code developers are aware that some of the information they see only needs to be remembered for a short period of time, while other information needs to be remembered over a longer period.

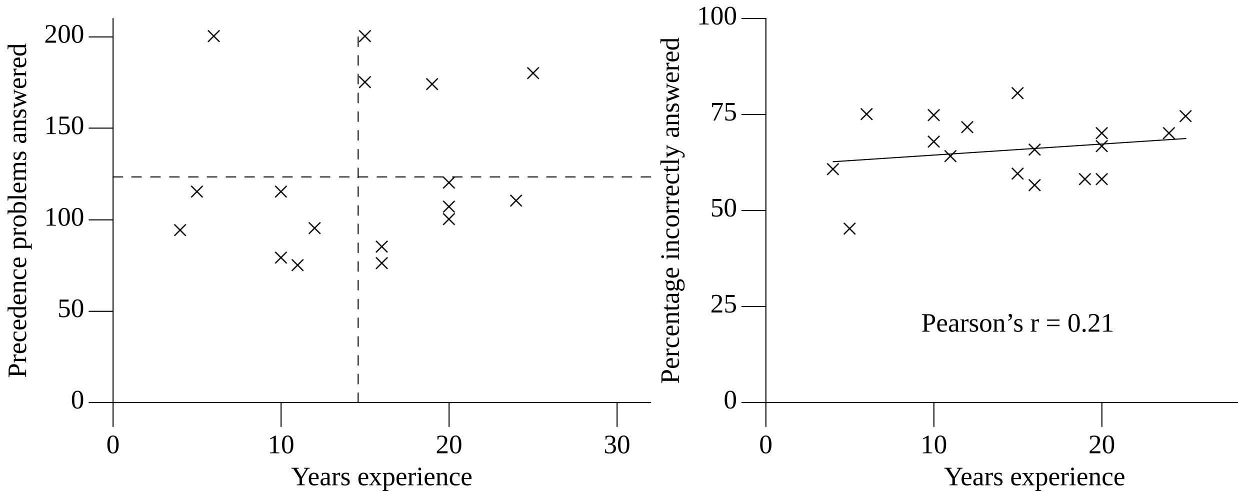

Figure .1: The left graph is the number of precedence problems answered (average 123.5) against number of years of professional experience for that subject (average 14.6). The right graph is the percentage of incorrect answers given by a subject against years of professional experience (the line is a least squares fit on the percentage of incorrect answers).

For instance, when deducing the affect of calling a given function the names of identifiers declared locally within it only have significance within that function and there is unlikely to be any need to recall information about them in other contexts. Each of the problems seen by subjects in this study could be treated in the same way as an individual function definition (i.e., it is necessary to remember particular identifiers and the values they represent, once a problem has been answered there is no longer any need to remember this information).

Subjects can approach the demands of answering the problems this study presents them with in a number of ways, including the following:

- seeing it as a challenge to accurately recall the assignment information (i.e., minimizing *would refer back* answers) and not being overly concerned about accurately answering the parenthesis questions,
- recognizing that *would refer back* is always an option and that it is more important to correctly parenthesize the list of expressions.
- making no conscious decision about how to approach the answering of problems,

Experience shows that many developers are competitive and that accurately recalling the assignment information, after parenthesizing the expression list, would be seen as the ideal performance to aim for.

At the end of the experiment subjects were asked to turn to the back page and list any strategies they used when answering problems. The answers given all related to remembering information about the assignment statements. There was no mention of the parenthesis problem. One person wrote on this page that: "Not usually programming in a language that supports shift operations." which suggests that they are using a language that is not among the set being considered here, and that the language they use assigns different precedences to the operators it supports (but then, whether the precedence actually defined by a language has any effect on developer performance is one of the questions raised by the results of this study).

6.3 Analysis of results

The 17 subjects gave a total of 2,100 answers to the operator precedence questions (average 123.5 per subject, standard deviation 44.0, lowest answered 75, highest answered 200). If subjects answered randomly then 50% of answers would be expected to be correct. In this experiment 66.7% of answers were correct (standard deviation 8.8, poorest performer 45.2% correct, best performer 80.5% correct).

If the hypothesis is correct, then subjects will give more correct answers for some operators than for others. A first approximation is given, for each operator, by the percentage of problems containing the operator that were answered correctly (see Table .5). It is an approximation because there are two operators in each problem and incorrect knowledge about one of the operators will affect impact the perceived performance for the other operator.

If subjects guessed all answers we would expect 50% to be correct. A correct percentage either side of 50% is evidence that subjects are consistently making use of some knowledge about the operator. When this knowledge agrees with that given in the language specification the correct percentage will be greater than 50% and when it is different the percentage will be less than 50%.

Some operators share the same precedence. If developers are aware of this, then it is possible that essentially the same piece of existing knowledge is reinforced in developer memory when either operator is encountered. If this is the case we would expect that subject performance for these *same precedence* operators will be similar. Table .5 shows that the percentage of correct answers for the operators `+` `-` and `==` `!=` `*` and `/` are very similar. Based on these results we might also expect subject performance for the *same precedence* operators `<<`, `>>` and `<`, `>`, `<=`, `>=` to be very similar, had they all appeared in problems.

The `%` operator has the same precedence as the `*` and `/` operators, but are developers aware of this connection? The very low percentage of correct answers containing `%` operator suggest that subjects were not aware of this fact.

The reinforcement that seems to take place when some same precedence operators appear in source means that an occurrence of either operator should count towards a learning experience for the same piece of knowledge. When plotting correct answers against operator occurrence, those operators which subjects know have the same precedence should have their occurrences summed (e.g., the 12.41% `+`'s and the 6.14% `-`'s are summed to give 18.55%).

Table .5: For each operator, the percentage of problems containing that operator that were answered correctly. Third column gives the total number of problems containing that operator.

Operator	Percentage correct	Number of problems
/	86.18	123
× 85.91	291	
||	78.89	379
&&	77.21	408
+	76.63	291
-	72.39	163
<	66.43	414
|	62.72	397
<<	60.19	417
==	60.14	291
!=	59.20	125
&	56.70	455
^	47.64	403
%	34.88	43

The percentage of correct answers for the `||` and `&&` operators was very similar (see Table .5). Do developers believe they share the same precedence level, or do they know the precedence is different (it actually differs by one level) and the performance similarity is caused by subjects randomly selecting one of the operators to have the high precedence? The percentage of correct answers for the `|` and `&` operators was not as similar (again the precedence levels differ by one). Answers to the questions raised by the results for these operators will have to wait until more experiments are performed.

In Figure .2 the Pearson r correlation coefficients are 0.64 and 0.27. There are 11 operators (or combination of operators), which gives 9 degrees of freedom. Looking in statistical tables[4] for a one-tail test we find that the results in the left plot are significant at the 0.05 level of significance, while the straight line fit of the

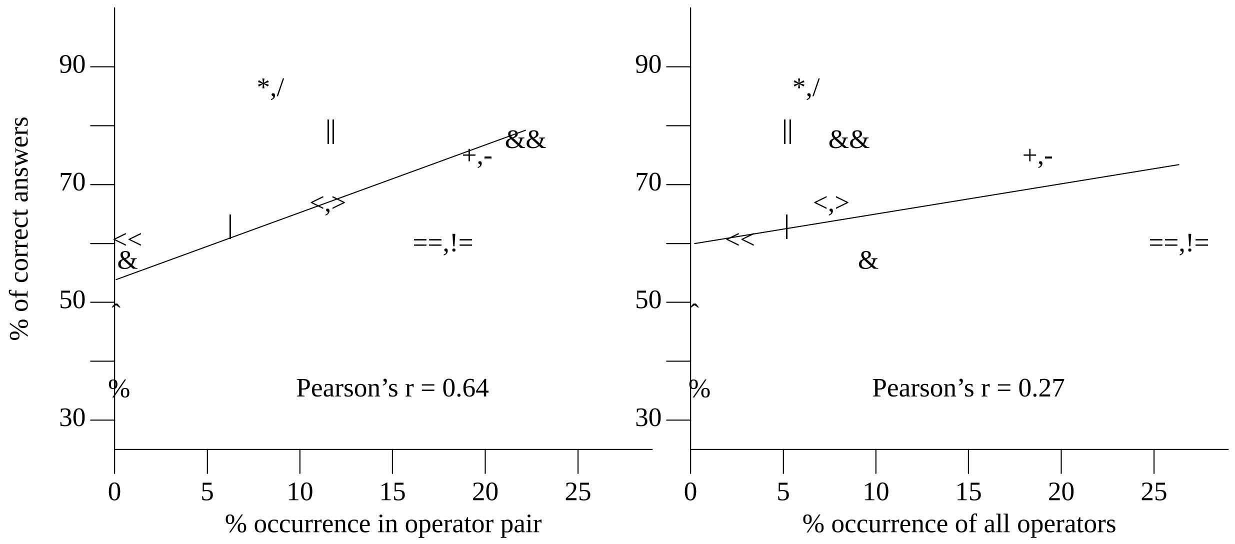

Figure .2: The left graph is a plot of the occurrence of operators, as a percentage of all operators in operator pairs, in visible C source (see Table .2), against percentage of correct answers to problems containing an instance of the operator (see Table .5). The line is a least squares fit assuming the percentage correct deviates from the regression line. Operators separated by a comma have had their percentage occurrences summed and the correct percentages averaged. The right graph plots operators as a percentage of all operators occurring in all expressions.

results in the right plot is not significant

Table .6: For each operator pair, the percentage of problems for which a correct answer was given for that operator pair. The value in parenthesis is the total number of problems containing that operator pair.

	×	/	%	+	-	<<	<	==	!=	&	^	|	&&	||
×	-													
/	90(10)	-												
%	-	-	-											
+	94(19)	69(13)	0(2)	-										
-	92(14)	100(1)	0(4)	80(10)	-									
<<	74(31)	84(13)	40(5)	58(31)	61(18)	-								
<	93(31)	100(13)	40(5)	90(33)	88(17)	64(50)	-							
==	100(19)	100(12)	-	100(24)	100(9)	83(31)	90(32)	-						
!=	100(12)	-	33(3)	100(6)	100(8)	92(13)	91(12)	57(7)	-					
&	71(35)	69(13)	40(5)	74(35)	47(21)	40(49)	41(51)	27(36)	11(17)	-				
^	63(30)	69(13)	25(4)	30(30)	33(15)	30(43)	16(43)	12(32)	8(12)	30(49)	-			
|	80(31)	84(13)	20(5)	70(30)	60(15)	45(46)	42(42)	22(31)	0(11)	79(44)	75(44)	-		
&&	100(31)	100(11)	60(5)	93(30)	94(18)	73(45)	75(44)	53(30)	71(14)	79(53)	88(45)	73(42)	-	
||	96(28)	100(11)	60(5)	92(28)	92(13)	69(42)	85(41)	53(28)	70(10)	87(47)	83(43)	86(43)	50(40)	-

6.4 Operator pair-wise analysis

Does each developer have their own, incorrect, model of operator precedence, or do many developers share a common incorrect model? For instance, do a significant number of developers believe that one particular operator has a much lower precedence than it does in reality?

The Bradley-Terry model is one technique for analysing results that consist of paired comparisons. A common example of the use of the Bradley-Terry model is in ranking sports teams from the results of individual matches played by those teams.

Table .7 gives the output of using the Bradley-Terry model to rank operators in precedence order based on subject answers. The second column for each operator can be interpreted as providing an estimate the probability that one operator will be chosen to have a higher precedence over another. For instance, the coefficient for the operator `^` is twice as great as the coefficient for `|` and so when both occur together in an operator pair the `^` operator is twice as likely to be assigned a higher precedence. However, be warned the error analysis is complex and the confidence in the calculated probabilities low; this issue is not discussed here.

The Bradley-Terry analysis was performed using the R system[8] with the addon package written by David Firth.[3]

Table .7: Coefficients obtained from applying a Bradley-Terry model to the pairs of binary operator precedence answers.

Operator	Coefficient	Operator	Coefficient
/	3.37	|	1.48
×	3.33	<	1.21
^	2.59	%	0.58
+	2.4	&&	0.15
-	2.15	==	0.09
&	1.87	||	0.05
<<	1.57	!=	0.0

The coefficients for the same precedence operators `*`, `/` and `+`, `-` and `==`, `!=` are very similar. However, there is little difference between the coefficients of many adjacently ranked operators. The rank assigned to the `%` operator is significantly different from the `*` and `/` operators (with which it shares the same precedence); this difference in precedence assignment is discussed later.

7 Discussion

The very large number of errors (approximately 33%) made by the subjects in this experiment is surprising. Had they guessed the answers the error rate would have been 50%. While experience shows that developers do make errors on the relative precedence of binary operators when writing and reading code, an error rate of this magnitude would generate many more faults (around 1% of all expressions would be in error) than appear to be encountered in practice.

Possible reasons for the discrepancy between the error rates found in this experiment and those seen in commercial software development include:

- Some aspect of the experiment's design resulted in subject's performance being significantly poorer than it is when they comprehend source code professionally. For instance, subjects may have been overly distracted by the desire to correctly remember information about the assignment statements (subject's performance on the 'filler' problem, an *`if-statement`* analysis task, in the 2004 experiment was so good that insufficient error data was obtained for analysis).
- Developers make use of local context (e.g., the semantics suggested by the names of variables), rather than knowledge of operator precedence, to deduce how operands bind to binary operators. For instance, the semantic associations of the identifiers used in the expression `num_apples % num_baskets > max_left_over` suggest that `num_apples` and `num_baskets` are in a calculation that checks whether there is an excessive number of apples remaining after some task is performed.
- Developers make use of knowledge of what is being calculated to choose the way in which operands bind to operators.
- Developers are aware of their own lack of knowledge of operator precedence and use parenthesis to indicate the intended precedence when they feel sufficiently unsure of the behavior that will occur. The ratios in Table .4 show that parenthesis are not always used for rarely occurring operator pairs. However, the high ratio of operator usage for some operators (e.g., see the `^` row) suggests that the overall impact may be significant.

Will measurements of binary operator usage in other language reveal different sets of common operator pairs? The most significant factor driving the choice of operators in source code is likely to be low level implementation details of the algorithms used. Various claims are made about how object oriented languages affect the choice and implementation of algorithms. Your author does not believe there is any significant operator usage variation between C++, C#, and Java. However, until detailed measurements are made it is not possible to claimed with any certainty that developer operator precedence performance for these languages is consistent with our hypothesis.

The percentage of correct answers for the `*` and `/` operators is well above that predicted by the least squares fit and it is also above that for the `+` and `-` operators. Of all the operators appearing in the problems the `*` and `/` operators have the highest precedence. Perhaps developers are aware of this *endpoint* information and it provides them with a decision algorithm that is simpler to use than for other operators, where information on an upper and lower precedence bound may need to be applied.

There is no obvious pattern to the other operators in the operator pairs that the multiplicative operators appeared with when a correct answer was given.

The percentage of incorrect answers for the `==` and `!=` operators is well above that predicted by the least squares fit. The Bradley-Terry analysis suggests that subjects believe, incorrectly, that the equality operators have the lowest precedence of all the operators seen in the study. While the results imply that subjects are using a much lower precedence for these operators than they actually have, the percentage of correct answers is not low enough to suggest that subjects are consistently using a lower precedence.

The results for the `^` operator are consistent with random guessing and this operator's precedence is sufficiently middling in the rankings that end-effects will be small.

Subject performance on the `%` operator is consistent with them believing it has a precedence level that is completely different than the one it actually has. More experiments are needed to uncover information on the model developers have for the `%` operator. These experiments might even inquire about the extent of developer knowledge about this operator (e.g., what operation to they believe it performs?).

Subject performance does not appear to be effected by the operator with which the `^` and `%` operators are paired.

7.1 Further work

What performance characteristics would we expect from subjects having much less experience than the subjects in this experiment? If subject performance is strongly correlated with operator pair reading experience, then we would not expect to see much less correlation between number of correct answers and source code occurrences.

Experiments using subjects who are just about to graduate and subjects a year or so after graduating (with and without extensive software development experience during that time) could provide the data needed to calculate the impact on performance of formal training and experience with source code.

A replication of the same experiment would be very useful, perhaps with some changes to the format. For instance, using different relational and shift operators, and more % problems; or changes the format of the problem to be solved so that a task other than remembering information about assignment statements is used.

What operations do developers think that the `^` and `%` operators perform? Perhaps the poor subject performance on these operators can be explained by the fact that many developers don't actually know what operation they perform, and hence where they might possibly fit in to the precedence hierarchy.

8 Conclusion

The results, Figure .2, are consistent with the hypothesis at the 0.05 level of significance. This means that there is a 5% chance that the null hypothesis (i.e., random answering) is true.

This is the first experimental evidence (perhaps partially indirectly) that software developer performance is effected by the number of times language constructs are encountered in source code.

If further experiments confirm the very poor binary operator precedence knowledge seen in this experiment, then developers must be using a significantly different technique for comprehending expressions (i.e., they are not relying on a knowledge of operator precedence).

Developer training is unlikely to be a viable option for solving the problem of faults caused by incorrect knowledge of operator pair precedence. While it is possible for people to learn the precedence of all language operators sufficiently well to pass a test, 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).

While coding guideline often contain a recommendation specifying that operator precedence be made explicit through the use of parenthesis, experience shows that developers are often loath to insert what they consider to be redundant parenthesis (because "*Everybody* knows what the precedence is and if they don't they should not be working on this code"). Subject operator pair performance should be all the evidence needed to convince people that redundant parenthesis do provide a useful service. Under 2% of all expressions would need to make use of redundant parenthesis.

9 Acknowledgments

The author wishes to thank everybody who volunteered their time to take part in the experiment and the ACCU for making a conference slot available in which to run it.

References

Citations added in version 1.0b start at 1449.

1. J. L. Elshoff. A numerical profile of commercial PL/I programs. *Software–Practice and Experience*, 6:505–525, 1976.
2. K. A. Ericsson, R. T. Krampe, and C. Tesch-Romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100:363–406, 1993. also University of Colorado, Technical Report #91-06.
3. D. Firth. Bradley-terry models in R. *Journal of Statistical Software*, 12(1):1–12, Jan. 2005.
4. P. R. Hinton. *Statistics Explained*. Routledge, 2nd edition, 2004.
5. D. M. Jones. I_mean_something_to_somebody. *C Vu*, 15(6):17–19, Dec. 2003.
6. D. M. Jones. Experimental data and scripts for short sequence of assignment statements study. http://www.knosof.co.uk/cbook/accu04.html, 2004.
7. D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1:105–133, 1971.
8. W. Venables, D. M. Smith, and R Development Core Team. An introduction to R. Apr. 2006.