Operand names influence operator precedence decisions (part 1 of 2)

Experiment performed at the 2007 ACCU Conference
First published in CVu vol. 20 no. 1

Derek M. Jones
derek@knosof.co.uk

1 Introduction

The 2006 ACCU experiment[3] threw up a surprising result; in 33% of cases experienced developers failed to correctly parenthesis an expression containing two binary operators to reflect the binding of operands to operator (i.e., the relative operator precedence). The result was surprising because if carried over into actual software development it would result in almost 1% of all expressions contained in source code[.1] being wrong (i.e., they would evaluate to a value different from that intended by the author, for instance the author or a subsequent reader of the expression `x & y == z` might incorrectly expect it to behave as if it had been written in the form `( x & y ) == z`).

Four possibilities spring to mind:

1. Subject's performance on the problem presented in the experiment is not a good approximation of their performance when writing code in real life.
2. When writing complicated expressions developers do make many precedence mistakes, but most of these are fixed before software goes into production use.
3. The source code measurements of parenthesis usage made in connection with the 2006 ACCU experiment (which show that developers often do not use parenthesis to specify the intended binding of operands to operators; see Table 3 & 4 of the write-up[3]) are not representative of other source code. The claim your author often hears in face-to-face discussion with developers is that they always make use of parenthesis to specify the intended precedence in complicated expressions.
4. When reading complicated expressions developers make use of additional information that is available within the context in which the expression occurs (e.g., what the expression is calculating, the names of the variables appearing in the expression, or the amount of spacing between operators and operands[4]).

The 2007 ACCU experiment was designed, among other things, to investigate one case involving the fourth of these possibilities.

This is the first of a two part article that reports on an experiment carried out during the 2007 ACCU conference investigating both impact of operand names on binary operator precedence selection and the consequences of a limited capacity short term memory on subject's performance in recalling information about assignment statements. This first article 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

Developers are often encouraged to use meaningful names for identifiers. The expectation is that meaningful names will provide information to subsequent readers of the code that will reduce the effort needed to comprehend that code. For instance, the name `flags` suggests an entity that denotes one or more on/off states.

The meaning associated with a name might also lead to an expectation about the kinds of operators used to manipulate variables having that name. For instance, an identifier having the name `flags` might be expected to appear much more often as the operand of a bitwise or logical operator than an arithmetic operator.

A name might have meaning to a developer because of its English language usage or because of being frequently encountered in a given context within source code. It is assumed here that presented with a problem involving source code, subjects will primarily make use of their experience of name usage in source code rather than any English language usage (i.e., occurrences in source code is used as a measure of context rather than occurrences in English prose).

Binary operators found in many programming languages might be divided into three broad categories.

1. Arithmetic operators. The operands of these operators can take on a range numeric values. Variables appearing in arithmetic operands might also appear as operands of relational or equality operators.

[.1]Only 1.9%of expressions in source code contain two or more binary operators.

2. Bitwise operators. The operands of these operators are treated as a sequence of bits. A common mapping is for a bit to denote some quantity having one of two values and a sequence of bits to denote a set of such distinct quantities. These operands might also appear in equality tests, but their semantics is such that they are unlikely to appear in relational tests.
3. Logical operators. The operands of these operators essentially have two possible values, zero and non-zero. The semantics of these operators makes it possible that they may appear in equality against a literal zero, but are unlikely to appear in an equality test against a variable operand (testing two variables for non-zero cannot be simply mapped to a single equality test). They are unlikely to occur in relational tests.

The hypothesis being tested by the parenthesis problem is that the names of identifiers, appearing as operands in an expression, can have a significant impact on subject's selection of operator precedence to be applied within that expression. For instance, when attempting to comprehend an expression such as `a + b & c` readers have to decide whether `b` is added to `a` or bitwise ANDed with `c`. This experiment manipulates the name of the operand that has two operators with which it could bind. Possible naming impacts include:

1. The name does not appear to provide any information that readers might use to help make a precedence decision.
2. The name is chosen such that readers might make use of their experience of identifier naming conventions, along with knowledge of English language usage, when making an operator precedence decision. This naming information can be experimentally manipulated to:
   - increase the probability that the correct precedence decision is made,
   - decrease the probability that the correct precedence decision is made (e.g., if `b` is replaced by `flags` then a reader may be influenced to believe (incorrectly) that `flags` is ANDed with `c` and the result added to `a`.

2.1 What meaning context might a name have?

A name can acquire meaningfulness to a person through repeated encounters in a given context. There are a number of situations in everyday life that require the use of arithmetic and binary concepts (e.g., *switch light on/off*, *change mind* and *flag an error*), and measurements confirm that words and phrases commonly used in human conversation are carried over into identifier names.

To maximise the effect that operand names have on subject's decision making the names need to be highly representative of the kinds of names that appear within source code in the various operator contexts. The selection was based on your authors intuition (i.e., they were made before any source code measurements were available). See Table .7 for information on the names chosen and their frequency of occurrence in the measured source code.

Table .1: Names and literals that your author believed, when creating the text of the problems, to most likely to occur as the operands of bitwise, logical and arithmetic operands. The last column contains names that are believed not to provide any information that can be used to help decide which operator context they might belong.

bitwise operands	logical operands	arithmetic operands	anonymous operands
error	mask	count	resource
flags	finished	num	val
state	flag	offset	field
bits	done	rate	n
options	started	length	temp
0xff	TRUE	2	7

There are situations where an identifier name belonging to one category might appear as the operand of an operator in another category. For instance, there are algorithms that use bitwise operators to more efficiently perform an arithmetic operation (e.g., shift left to multiply by a power of 2), and it is possible that an arithmetic variable appears as the operand of a logical operator because the author omitted an explicit equality test against zero (because the language specifies that an implicit test is performed).

It is straightforward to assign a context to many of the binary operators (see Table .2). The only context that makes sense for the relational operators is an arithmetic one (source code measurements showed the same names appearing frequently as the operands of both kinds of operator). While there might be situations where meaning can be assigned to a relational comparison of bitwise or logical operands, these are likely to be rare. The equality operators might be applied to operands having any context and are classified as anonymous operators, which means that they do not provide any information that might be used to decide whether a particular operand binds to them or another operator.

Table .2: Binary operators and their associated context. The left operand of a shift operator has a bitwise context, while the right operand has an arithmetic context and the result has a bitwise context.

bitwise operator context	logical operator context	arithmetic operator context	anonymous operator context
& | ^	&& ||	× / %	== !=
		+ -	
		< > >= <=	
<< >>		<< >>	

2.2 Related work

Most of the operator/operand categorization techniques place restrictions on what are considered to be legitimate combination of pairs of operands, or operator/operand combinations. Possibilities include:

- most languages do not support the unrestricted mixing of operands of different types in an expression. For instance, arithmetic operations cannot be performed between an integer and string (some languages attempt to support this by performing implicit conversions). There are languages that support sophisticated typing mechanisms which allow developers to create new types which cannot be mixed with operands of a different type.
- in scientific calculations, dimensional analysis restricts the mathematical operations that can be performed on values denoting some physical quantity (because no physical meaning can be given to the operation). For instance, adding a distance to a time value has no meaning while dividing distance by time yields the velocity. Tools that check scientific software for consistent use of physical dimensions are available.[5]

The names of the operands are not part of these categorization techniques. Information on these restrictions is explicit specified as part of a language or application specification and not something that a developer might subconsciously pick up over time.

In a study by Bassok, Chase and Martin[1] subjects were asked to create mathematical problems involving named everyday objects. The results showed that subjects often made use of semantic relationships between the named objects. For instance, given apples and baskets the problems generated by subjects might involve dividing the number of apples by baskets, but rarely involved adding them.

3 Measuring operand name usage

This subsection discusses measurement of names appearing as or in the operands of some binary operators within 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.

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

When the operand was a function call the identifier denoting the function was used, when the operand was an array access the identifier denoting the array was used, when the operand was a structure or union member selection the identifier denoting the member was used (e.g., in `p.q` the identifier of interest was taken to be `q`).

Operands whose value was obtained by applying the indirection operator (i.e., unary `*`) were not counted.

A variety of conventions are used to create the character sequence in an identifier name. Developers extract information from identifiers by recognising character sequences that have meaning to them. These character sequences might be words, abbreviated words, acronyms or sequences of two or more of these.

There are three common conventions for explicitly highlighting the boundary between two character sequences:

1. Use of the underscore character (i.e., _). For instance, `mumble_flags`.
2. Writing the first character of a new sequence in upper case. For instance, `mumbleFlags`.
3. Relying on reader expertise to work out where the character sequence boundary exists. For instance, `mumbleflags`.

These measurements applied the first two conventions to extract names from identifiers. In the case of the second point above: any sequence of digit characters was treated as a break-point between two names; and a run of two or more upper case letters was treated as a single name (i.e., the last upper case letter was not regarded as being the start of a name using the second convention given above).

An identifier name created by joining together two or more words might be viewed as part of a sentence or a compound word. For instance, `mumble_flags` might be interpreted as two or more flags associated with mumble (and probably occurring in a bitwise or logical context), while `num_flags` might be interpreted as an abbreviated form of the phrase *number of flags* (and probably occurring in an arithmetic context). These measurements do not attempt to deduce the interaction between any words appearing in an identifier considered to contain two or more words.

Table .3: The top 10 most common names appearing as or in identifiers as the operands of binary operators in the visible source of `.c` files. *Programs* the number of programs containing an instance of the identifier having or containing the name, *Occurrences* the number of occurrences of the name, *Arithmetic* the percentage of occurrences in an arithmetic context, *Bitwise* the percentage of occurrences in a bitwise context, *Logical* the percentage of occurrences in a logical context, *Equality* the percentage of occurrences as the operand of an equality operator.

Programs	Occurrences	Name	Arithmetic	Bitwise	Logical	Equality
7	35627	i	86.3	5.7	0.4	7.7
7	25201	null	0.1	0.1	0.1	99.7
7	14821	size	79.9	7.7	0.4	12.0
7	14121	type	8.3	8.4	2.4	81.0
7	12160	flags	0.5	97.4	0.2	1.9
7	12091	len	87.3	3.7	0.5	8.5
7	9396	count	78.2	4.7	1.2	15.9
7	9345	max	88.0	2.4	0.2	9.4
7	8668	code	7.0	3.8	0.6	88.6
7	8462	status	16.9	56.9	0.5	25.7

There were 136,127 identifiers that appeared as the operand of one of the binary operators of interest in this study, from which 30,683 unique names were extracted. Table .3 lists the 10 most commonly seen names, Table .4 the 10 most commonly seen names in a bitwise context and Table .5 the 10 most commonly seen names in a logical context (in this case the majority of *all* occurrences requirement was relaxed to the requirement that the percentage occurrence in a logical context be greater than any other context).

An application domain specific name might be frequently used in one program but not in any other, e.g., `inb` in Table .4. Unless a developer has experience working on that program they might not be aware that in some domains it has a common usage in a given context. The *Programs* column lists the number of different programs containing at least one instance of a name.

Table .4: The top 10 names appearing with greater that 50% frequency as or in identifiers as the operands of bitwise binary operators in the visible source of `.c` files.

Programs	Occurrences	Name	Arithmetic	Bitwise	Logical	Equality
7	12160	flags	0.5	97.4	0.2	1.9
7	8462	status	16.9	56.9	0.5	25.7
7	5854	mask	6.4	89.0	1.1	3.6
7	4815	read	20.6	50.8	1.5	27.0
7	4162	f	25.0	53.5	3.1	18.3
7	2827	val	28.7	50.8	0.8	19.6
7	2270	stat	14.5	71.0	0.8	13.7
7	2322	flag	3.8	70.6	6.4	19.3
1	1764	inb	2.2	83.7	0.1	14.1
7	1346	active	9.6	50.7	4.2	35.5

Table .5: The top 10 most common names, where the percentage occurrence in a logical context is greater than any other context, appearing as or in identifiers as the operands of logical binary operators in the visible source of `.c` files.

Programs	Occurrences	Name	Arithmetic	Bitwise	Logical	Equality
7	2631	is	2.4	18.4	55.5	23.7
6	963	user	20.6	9.6	38.8	31.0
4	329	operand	21.6	20.1	27.4	31.0
5	280	constant	15.0	15.7	26.4	42.9
3	251	reload	16.7	6.4	31.1	45.8
5	227	template	13.2	0.4	26.0	60.4
3	213	charset	10.3	4.2	36.2	49.3
3	191	put	22.5	11.5	59.7	6.3
7	182	after	9.3	13.7	26.9	50.0
3	165	pthread	5.5	0.0	12.7	81.8

Given the results of the 2006 experiment (i.e., 33% of answers given by subjects were incorrect) measurements based on the assumption that the author of the expression `a + b & c` expected `b` to be the operand of an arithmetic operation could be significantly inaccurate. The operand name measurements given here are restricted to those subexpressions where the intended binding is considered to be unambiguous. For instance, when brackets delimited a single binary operation (e.g., `(a + b) & c` or when an expression contained a single binary operator (the simple assignment or compound assignment operators were not treated as binary operators for the purposes of these measurements). This restriction did not have a significant impact on the amount of data obtained. As Table .6 shows most expressions contain either zero or one of the binary operators of interest here.

Table .6: 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

3.1 Occurrence of names used in the problems

Table .7 lists the number of occurrences of each name used in the parenthesis problems along with the percentage occurrence as the identifier (or part of an identifier) in the operand of arithmetic, bitwise, logical or equality operators. The measurements show that your author's intuition is not always a reliable indicator of what constitutes common usage of names in some contexts.

Table .7: Actual occurrences, in the visible `.c` files, of the names used (Table .1) in the parenthesis problem. The *Other operands* entry is included to provide supporting information on `TRUE` context usage. See Table .3 for a description of the columns.

Name	Programs	Occurrences	Arithmetic	Bitwise	Logical	Equality
arithmetic operands						
count	7	9396	78.2	4.7	1.2	15.9
num	7	7905	81.1	3.3	0.5	15.1
offset	7	5889	79.5	10.9	0.8	8.9
rate	5	528	62.9	17.6	0.4	19.1
length	7	5097	77.4	6.5	0.4	15.7
2						
bitwise operands						
error	7	2960	12.1	21.8	1.4	64.7
flags	7	12160	0.5	97.4	0.2	1.9
state	7	5723	5.6	29.2	0.8	64.4
bits	7	2934	59.3	27.5	0.2	12.9
options	6	543	13.4	74.6	2.8	9.2
0xff						
logical operands						
mask	7	5854	6.4	89.0	1.1	3.6
finished	6	50	46.0	8.0	0.0	46.0
flag	7	2322	3.8	70.6	6.4	19.3
done	7	578	14.5	29.9	4.5	51.0
started	4	91	16.5	61.5	2.2	19.8
TRUE	7	735	2.7	0.0	0.7	96.6
anonymous operands						
resource	3	481	33.9	15.8	0.0	50.3
val	7	2827	28.7	50.8	0.8	19.6
field	7	523	18.7	32.3	4.2	44.7
n	7	5141	78.8	6.4	0.6	14.1
temp	7	1588	30.5	45.0	1.0	23.5
7						
Other operands						
FALSE	7	1046	0.0	0.1	0.4	99.5

- **Arithmetic operands**. All of the names used in the arithmetic context problems occurred much more frequently in this context than any other context.
- **Bitwise operands**. Two of the names used in the bitwise context problems occurred much more frequently in this context than any other context and two occurred just over twice as frequently in this context as an arithmetic context. One name `bits` appeared most often in an arithmetic context, although the singular form (i.e., `bit`) occurred more frequently in a bitwise context (see Table .8).

- **Logical operands**. All of the names used in the logical context problems occurred much more frequently in other contexts. The name TRUE was actually more common in an arithmetic context and less common in a bitwise context. Using this name as an operand was probably a mistake because in a real logical operand context within source it would effectively create dead code. However, it did occur in the problems seen by subjects and given that the corresponding name FALSE occurred most commonly in a logical operand context, this name maintained its logical context classification. The semantics of the logical operators (for the languages of interest here) specifies that each operand is implicitly compared against zero. Because of the implicit comparison there is a common developer practice of omitting an explicit comparison. For this common usage case the operands of the logical operators might be expected to have names that imply an arithmetic or bitwise usage. The source code measurements in Table .5 bear this out. Names appearing in the operand of a logical operator also frequently occur in operands of arithmetic and bitwise operators. As Table .2 in the 2006 article[3] shows, the logical operators occur sufficiently often in source code that if operator context specific names were commonly used they would show up in the source measurement counts.

- **Anonymous operands**. There is no consistent pattern of usage., with some occurring twice as frequently in one context as another. By far the most common occurrence of the name n was in an arithmetic context and its use was reclassified as such.

Table .8: Actual occurrences, in the visible of .c files, of the names used (Table .1) in the parenthesis problem. See Table .3 for a description of the columns.

Name	Programs	Occurrences	Variant	Arithmetic	Bitwise	Logical	Equality
error							
	7	2013	err	21.2	31.1	0.4	47.2
	3	175	errors	54.3	20.6	0.6	24.6
flag							
	2	1024	cflag	0.0	90.5	1.1	8.4
state							
	5	291	states	6.9	89.7	0.7	2.7
bits							
	7	2399	bit	35.0	48.2	4.5	12.3
	4	200	bitmap	42.5	15.0	1.0	41.5
options							
	6	453	option	16.6	48.3	3.5	31.6
finished							
	5	81	finish	27.2	1.2	3.7	67.9
started							
	7	4565	start	76.2	10.6	0.6	12.7
count							
	6	305	counter	79.0	7.2	0.0	13.8
num							
	7	1922	number	56.7	9.9	0.4	33.0
offset							
	7	1255	off	69.3	13.9	1.6	15.2
	2	137	offs	92.0	0.0	0.0	8.0
length							
	7	12091	len	87.3	3.7	0.5	8.5
val							
	7	4006	value	36.3	24.4	3.8	35.4
	7	492	retval	43.9	13.6	0.0	42.5
field							
	5	90	fields	55.6	0.0	3.3	41.1
temp							
	7	2223	tmp	49.8	27.8	0.8	21.7

Some names appearing in the source might be considered to be variant spellings of a name appearing in a problem. Table .8 lists various character sequences found in the source which might be considered to denote the same name, along with number of occurrence information.
Some of the possible variant spellings appearing in Table .8 are actually more common than the names actually used (e.g., compare `value` vs. `val` in Table .7). Any future experiments based on these problems might like to use the more common variant name.
When analysing the results the names were reclassified according to the findings of the source code measurements (see Table .9).

4 The Experiment

The experiment was run by your author during a 45 minute lunchtime session of the 2007 ACCU conference (http://www.accu.org) held in Oxford, UK. Approximately 290 people attended the conference, 6 (2%) 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.

4.1 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.[2]
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 ----------------------
 propeller =  6;
 sofa =  5;
 chair =  4;
--------------------- second side of sheet starts here ---------------------
Oxff  |  rate  *  num
field  <  rate  ^  flags
error  &  2  <  count
done  ||  finished  |  OXFF
field  !=  7  ^  started
              remember    would refer back     not seen
 chair     =    ____          __                __
 table     =    ____          __                __
 sofa      =    ____          __                __
 propeller =    ____          __                __
```

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 keep track of 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).

4.2 The parenthesis problems

Based on the results of the 2006 ACCU experiment[3] it was anticipated that on average subjects would answer around 20 complete questions. This gave an estimated 100 answers per subject (subjects actually averaged 123.5 answers).

The selection of operands and operators followed the same procedure as the 2006 experiment, with the following exceptions (using the name contexts given in Table .1 and operator contexts given in Table .2):

- The name of the first operand was chosen from the set of names considered to have the same context as the first operand and similarly for the name of the third operand with respect to the second operator.
- The name of the second operand (i.e., the one that might bind to either operator) was selected (randomly with equal probability) from either the set of names having the same context as the left operand or the set of names having the same context as the right operand.
- The name of the middle operand was never the same as that of the first or third operand.
- There was a 10% probability that the letters of any operand were converted to upper case.

5 Results

Source code measurements showed that the names originally used in the creation of experimental problems (Table .1) did not always occur in the expected context. The findings of the source code measurements were used to reclassify some names as belonging to a different context, see Table .9.

Table .9: Names and literals with their associated context, recategorised based on measurements of C source code.

bitwise operands	logical operands	arithmetic operands	anonymous operands
error		count	resource
flags		num	val
state		offset	field
bits		rate	temp
options		length	7
0xff	TRUE	n	2
mask			
flag			
finished			
done			
started			

5.1 Subject experience

The average subject experience was 14.5 years (standard deviation 9.0). Because of the small number of subjects it is not possible to draw any statistically significant correlations between subject experience and other quantities.

Fortunately C, C++, C#, Java, Perl, and many other languages use the same relative precedence for those operators used in this experiment, so there is no need to worry about the possibility of subjects (attending the Association of C and C++ Users conference) getting confused about which language an expression might be written in.

5.2 Analysis of results

The 6 subjects gave a total of 697 answers to the operator precedence questions, an average 116.2 per subject (standard deviation 35.0, with 123.5 in 2006. The lowest number of problems answered was 75, highest number 174.

If subjects answered randomly the total number of answers expected to be correct would have a Binomial distribution. With such a distribution there is a 15.9% probability that more than 55% of answers would be correct, 2.3% probability that more than 60% of answers would be correct, and a 0.1% probability that more than 65% of answers would be correct

In this experiment 65.3% (standard deviation 8.7) of answers were correct (66.7 in 2006), poorest performer 51.2% correct (45.2 in 2006), best performer 77.8% correct (80.5 in 2006).

If the results support the hypothesis, then in some cases the name of the operand will have a significant impact on the percentage of correct answers given. There are two main possibilities, one of which has four combinations:

1. Both of the operators in the expression seen by a subject have the same context. In this case the name of the operand does not provide any additional information to subjects (or at least those used in these problems do not). Table .10 shows that this occurred in 114 (16.4%) answers, with subjects being correct in 76.3% of cases (356 answers with 71.3% correct in 2006) This is higher than the 65.3% average and while it suggests that subjects might have more accurate knowledge of the relative precedence of operators that belong to the same context, but Table .11 tells a different story.
2. The operators do not share the same context and:
   (a) the operand has the same context as the operator with the highest precedence (i.e., the operator to which the operand will bind). This occurs in 251 answered problems and the answer was correct in 72.5% of cases.
   (b) the name of the operand has the same context as the operator to which precedence does not bind it. This occurs in 129 answered problems and the answer was correct in 43.4% of cases.
   (c) the name of the operand does not have the same context as either operator. In this case the name of the operand does not provide any additional information to subjects (or at least those used in these problems do not). This occurs in 177 answered problems and the answer was correct in 64.4% of cases.
   (d) the name of the operand has the same context as both operators (this can only occur for a shift operator). In this case the name of the operand does not provide any additional information to subjects (or at least those used in these problems do not). This occurs in 26 answered problems and the answer was correct in 61.5% of cases.

It is possible for the identifier TRUE to be treated as having a bitwise or logical context (It was originally assigned to the logical context). Analysis of the answers show that the results are very similar whichever context it is treated as having.

Table .10: Numbers of answers for various combinations of operator/operand context. *Match higher* occurs when the operand context matches the context of the operator with the high precedence (i.e., the one to which it binds). *Match lower* occurs when the operand context matches the context of the operator with the lower precedence (i.e., the one to which it does not bind). For example, *Match higher/Not match lower* refers to the case where the operand context matches that of the operand with the higher precedence and does not match the context of the operand with the lower precedence.

Operator/operand context match status	Total answers	% correct	% wrong
Both operators have same context	114	76.3	23.7
Match higher/Not match lower	251	72.5	27.5
Match higher/match lower	26	61.5	38.5
Not match higher/Not match lower	177	64.4	35.6
Not match higher/match lower	129	43.4	56.6

The combination *Match higher/match lower* only occurs when of the operators is a shift operator.
The combination *Not match higher/Not match lower* only occurred when of the operators was either a relational, equality or a shift operator.
Table .10 suggests (first row) that subject performance is much better when the precedence decision involves operators having the same context compared to when the operators come from different contexts. However, a break-down by operator context (see Table .11) shows that this difference is primarily attributable to significantly better performance with arithmetic operators.

Table .11: Break-down of percentage of correct answers, by operator context and year of experiment, when both operators have the same context. Value in parenthesis is the total number of answers.

Year	Arithmetic	Bitwise	Logical
2007	96.2 (53)	56.5 (46)	58.3 (12)
2006	85.5 (172)	60.6 (137)	50.0 (40)

6 Discussion

Source code measurements confirmed that distinct operand names are commonly used for arithmetic (Table .3) and bitwise operators (Table .4). However there was no obvious distinct naming used for the operands of logical operators (Table .5).
Subject performance was significantly better when both operators were arithmetic. Arithmetic operators occur together much more often in source code, compared to bitwise operators occurring together and logical operators occurring together (see Table 3 of the 2006 article[3]). Subjects who had followed a numerate academic path will also have significantly more experience with arithmetic operators than other operators. So significantly better performance when both operators are arithmetic is to be expected.
The results, Table .10, show that the name of the operand can significantly increase or decrease the number of correct answers, depending on whether the name context is the same as the operator to which it does or does not bind. Compared to an approximate 64% correct answer rate when the name context matches both or neither operator contexts the percentage of correct answers either increased to 72.5% or decreased to 43.3%.
The use of numeric values as operands introduced unnecessary complications for little benefit. If you author had the chance to rerun the experiment they would probably be removed.

6.1 Further work

Beginner programmers are often told to use meaningful names, often without being told what constitutes a meaningful name. Naming conventions are therefore something that has to be picked up by experience. 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.
Within an expression containing only arithmetic operators (or an expression containing only bitwise operators), do developers make use of operand name information to make precedence decisions? For instance, the study

by Bassok, Chase and Martin[1] found that subjects often used some name combinations (e.g., apples and baskets) prompted a division operation.

This experiment used English language names. What operand names are commonly associated with arithmetic and bitwise contexts in other human languages? Are developers who speak English as a second language as sensitive to naming context as native English speakers?

A replication of the same experiment would be very useful, perhaps with some changes to the format. Also measurements of source code written in other computer languages might find context differences in operand naming and operator usage.

7 Conclusion

The hypothesis that developers make use of context information contained in identifier names, when deciding which operator an operand binds to, was supported by the results of this experiment. The impact of naming information on precedence decision making was significant.

Subject precedence performance to significantly better when an expression only contained arithmetic operators, compared to when it contained operators from other contexts (mixed or other context).

Source code measurements showed that distinct operand naming only occurs for arithmetic and bitwise operators and that there was no obvious distinct naming for logical operators.

8 Acknowledgements

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. Thanks to Les Hatton for comments on an early draft.

References

1. M. Bassok, V. M. Chase, and S. A. Martin. Adding apples and oranges: Alignment of semantic and formal knowledge. *Cognitive Psychology*, 35(2):99–134, Mar. 1998.
2. D. M. Jones. Experimental data and scripts for short sequence of assignment statements study. http://www.knosof.co.uk/cbook/accu04.html, 2004.
3. D. M. Jones. Experimental data and scripts for developer beliefs about binary operator precedence. http://www.knosof.co.uk/cbook/accu06.html, 2006.
4. D. Landy and R. L. Goldstone. The alignment of ordering and space in arithmetic computation. In *Proceedings of the Twenty-Ninth Annual Meeting of the Cognitive Science Society*, pages 437–442, Aug. 2007.
5. G. W. Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software–Practice and Experience*, 31(11):1067–1076, Sept. 2001.