# The New C Standard (Excerpted material)

## An Economic and Cultural Commentary

**Derek M. Jones**
derek@knosof.co.uk

## 6.6 Constant expressions

constant expression syntax

1322

```
constant-expression:
                conditional-expression
```

### Commentary

constant expression not contain 1324

Syntactically specifying a *constant-expression* as a *conditional-expression* is consistent with constraints on operators that may appear in such an expression.

### Coding Guidelines

How do people perform simple arithmetic? For operations involving small single-digit values, current models of human performance[2, 10] are based on a network of connected arithmetic facts (e.g., $1 + 1 = 2$) stored in long-term memory. These *facts* are accessed by a process of spreading activation,[6, 7] using as input the numbers and operators provided by the calculation. This process does not appear to require a person to have any understanding of the operation performed;[5] it is purely a fact-retrieval operation. People tend to use arithmetic procedures for larger values. The extent to which either a retrieval or procedural approach is used has been found to vary between cultures.[3] For a discussion of how people store arithmetic *facts* in memory, see Whalen.[16] For a readable introduction to human processing of simple arithmetic quantities, see Dehaene.[4]

There have been some studies investigating people's eye movements while they performed simple arithmetic tasks.[13] However, the operands in source code expressions are written on a horizontal line, not vertically under each other (as we are taught to do it in school). It is not known how this difference in operand layout will affect the sequence of eye movements that occur when performing simple arithmetic with source code expressions.

When a sequence of simple arithmetic operations are performed, the order in which they occur can affect the response time and error rate.[1] Other performance related behaviors include:

- *Problem size/difficulty.* Both the time taken to produce an answer and the error rate increase as the numeric value of the operands increases. It is thought that this effect occurs because operations on larger values are not as common as on smaller values; it is an amount-of-practice effect. Constants in C source code also tend to be small (see Figure **??**).
- *Split effect.* People take longer to reject false answers that are close to the correct answer (e.g., $4\times7 = 29$) than those that are further away ($4\times7 = 33$).
- *Associative confusion effect.* Here people answer a different question from the one asked[8] (e.g., giving 12 as the answer to $4\times8 =?$, which would be true had the operation been addition).
- *Odd/even effect.*[11] Here people use a rule rather than retrieving a fact from memory to verify the answer to a question; for instance, adding an odd number to an even number produces an odd result.

developer errors memory overflow

The affect of working memory capacity limits on mental arithmetic performance is discussed elsewhere.

### Description

A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be. 1323

### Commentary

footnote 96 1334

The operative phrase here is *can be*. The standard specifies a number of contexts where a numeric value is required at translation time. In all other cases a translator need only act as if the expressions are evaluated during translation. The standard also gives implementations freedom to evaluate other expressions at translation time. There is no abstract machine state available during translation, which limits what can be

constant expression other forms 1344

considered to be a constant expression.

Here the term *constant expression* refers to the syntactic form `constant-expression`, while *constant* refers to the syntactic form `constant`. Evaluation of constant expressions, by a translator, is often called *constant folding* (particularly in compiler writing circles).

constant syntax

### C++

The C++ Standard says nothing about when constant expressions can be evaluated. It suggests (5,19p1) places where such constant expressions can be used. It has proved possible to write C++ source that require translators to calculate relatively complicated functions. The following example, from Veldhuizen,[14] implements the pow library function at translation time.

```
template<int I, int Y>
struct ctime_pow
   {
   static const int result = X * ctime_pow<X, Y-1>::result;
   };

// Base case to terminate recursion
template<int I>
struct ctime_pow<X, 0>
   {
   static const int results =- 1;
   };

const int x = ctime_pow<5, 3>::result; // assign five cubed to x
```

### Other Languages

Some languages do not support the use of operators in constant expressions; which are essentially literals. For instance, the first standard for Pascal, ISO 7185, did not; many implementations added such support as an extension and the first revision of that language standard added support for such usage. Other languages (e.g., Ada) allow the value of a constant to be calculated during program execution.

### Common Implementations

There are two ways of representing integer constants during translation. A translator can either use the representation of the host on which the translator is executing, or it can use its own internal format (potentially often supporting a greater range of values). The latter approach has the advantage of offering consistent behavior, a benefit for those vendors offering products on a variety of hosts.

There is no context where the standard requires a translator to support arithmetic operations on floating constants during translation. Many implementations choose to generate code to evaluate expressions containing floating-point operands during program startup

Some translators and static analysis tools issue diagnostics for operations they consider to be suspicious; for instance, a left-shift operation that results in all of the bits set in the left operand being shifted out of the result (which can occur for a shift amount less than the width of the promoted left operand).

Flow analysis is used by some translators to deduce that particular uses of objects have a single value (known as *constant propagation*). For instance, if x appears in an expression immediately after the value 3 is assigned to it, a read of x can be replaced by 3. Deducing whether an expression has a unique constant value during program execution is undecidable[9,12] (even if the interpretation of conditional branches is ignored). Rather than deducing a unique constant value, *generalized constant propagation* attempts to estimate the range of values an object may have at a particular point in the source. Verbrugge, Co, and Hendren[15] compare the performance of various algorithms on C programs of under 1,000 statements.

### Example

```
extern int glob;

```

```
3  void f(void)
4  {
5  glob = 4;
6
7  struct {
8          /*
9           * It may be possible to deduce the value of glob at translation
10          * time, but the standard does not require such deduction.
11          */
12         int mem :glob; /* Constraint violation. */
13        } loc;
14 glob++; /* Can be implemented by assigning 5 to glob. */
15 }
```

### Constraints

constant expression not contain

Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[96)] 1324

#### Commentary

constant expression 1322 syntax

While the syntax does not permit a constant expression to contain an assignment or comma operator at the outermost level, it does not prohibit them from occurring within a parenthesized expression. This constraint specifies these operators cannot occur in any contexts that are evaluated within a constant expression. Because they are evaluated at translation time, an attempt to generate a side effect in a constant expression, could not be interpreted to have any meaning. The C abstract machine does not exist during translation.

abstract machine C

#### C90

*. . . they are contained within the operand of a* ***sizeof*** *operator.*[53)]

sizeof result of

With the introduction of VLAs in C99 the result of the **sizeof** operator is no longer always a constant expression. The generalization of the wording to include any subexpression that is not evaluated means that nonconstant subexpressions can appear as operands to other operators (the logical-AND, logical-OR, and conditional operators). For instance, `0 || f()` can be treated as a constant expression. In C90 this expression, occurring in a context requiring a constant, would have been a constraint violation.

#### C++

Like C90, the C++ Standard only permits these operators to occur as the operand of a **sizeof** operator. See C90 difference.

#### Other Languages

Languages that support constant expressions that are evaluated during program execution may also support the occurrence of side effects.

#### Coding Guidelines

redundant code

As footnote 96 points out, it is possible for a subexpression not to be evaluated and still affect to the final value of the expression that contains it. A subexpression that does not affect the final value of the expression is not redundant code in the sense that it does not form part of the program image. An expression containing identifiers that are expanded by the preprocessor may have a generalized form that has been designed to handle a variety of different translation and host environments.

In the following example, a particular constant may only be available when translating for a particular host. The generalized case involves a function call during program execution. A macro definition hides some implementation details behind the name `Y`; however, if it needs to be used in a context where a constant is

required, more complexity may be needed. In this case another macro, X, and a conditional expression has been used.

```
#if defined VENDOR_A
   #define X 0
   #define Y VENDOR_A_VALUE
#else
   #define X NON_ZERO_DEFAULT_VALUE
   #define Y get_value_during_execution()
#endif

int glob[X ? X : Y];
```

Although a subexpression that does not affect the final value of an expression and does not contain identifiers that are macro expanded involves a cost and probably has no benefit; such uses are rare. For this reason no guideline recommending against using such constructs is given.

**Example**

```
extern int glob;
extern int g(void);

int f(void)
{
switch (glob)
   {
   case 0 && g() : return 77;

   case 1 || g() : return 88;

   case 1 ? 2 : g() : return 99;

   case sizeof(g()) : return 111;
   }
}
```

---

1325 Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

**Commentary**

If the evaluation of an expression, at execution time, would deliver a result that is not representable in its type, the behavior is undefined. The check on constant expressions can afford to be more rigorous because there is no execution-time penalty in translation-time checks on the evaluation of a constant expression.

exception condition

**C++**

The C++ Standard does not explicitly specify an equivalent requirement.

**Example**

```
#include <limits.h>

int a[INT_MAX * INT_MAX];
```

## Semantics

---

1326 An expression that evaluates to a constant is required in several contexts.

### Commentary

footnote 96 1334
designator
constant-expression
conditional inclusion
constant expression
initializer
static storage duration object

The standard lists many of the contexts requiring integer constant expressions (additional ones include the *constant-expression* in a designator and the expression that controls conditional inclusion). All initializers for objects having static storage duration must be constant.

### Other Languages

A few languages do not require constants in any contexts. However, the execution-time overhead on such a language design decision is often perceived to be much higher than users are willing to tolerate. It also complicates the job of writing a translator.

### Common Implementations

Extensions created by vendors sometimes relax the contexts in which a constant expression is required. A common extension is to allow nonconstant expressions in initializers for objects having file scope (also supported in C++).

### Coding Guidelines

controlling expression
if statement

As discussed elsewhere, expressions that evaluate to constants in other contexts are sometimes suspicious.

---

If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment. 1327

### Commentary

floating constant
conversion
not raise exception

This is a requirement on the implementation. The prohibition against raising exceptions applies to floating constants, not to constant expressions.

To ensure adequate range and precision, a translator that evaluates floating expressions will need to be aware of the value of the FLT_EVAL_METHOD macro. Arithmetic range and precision are two attributes of floating-point expression evaluation. Others, not included by this requirement, are specified by the FLT_ROUNDS macro and the FLT_RADIX macro. Also, while the status of FENV_ACCESS macro is known to the implementation, the affect of any execution-time calls to the floating-point status library functions will not.

FLT_EVAL_METHOD
FLT_ROUNDS
FLT_RADIX

### C++

This requirement is not explicitly specified in the C++ Standard.

### Other Languages

Very few languages require floating expressions to be evaluated during translation.

### Common Implementations

A few implementations (e.g., gcc) do evaluate floating expressions during translation. Translators may, or may not, provide an option to control the optimization of floating-point operations (gcc provides many).

---

integer constant expression

An *integer constant expression*[97] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, and floating constants that are the immediate operands of casts. 1328

### Commentary

This defines the term *integer constant expression*.

The restriction on floating constants only being the *immediate operands of casts* means that translators are not required to perform arithmetic operations on floating constants. Evaluating the result of casting of a floating-point constant to an integer type can be achieved by manipulating sequences of characters, starting with the initial token (no numeric representation of the floating constant need be created).

For the result of the **sizeof** operator to be an integer constant, its' operand cannot have a VLA type.

**C++**

> . . . , **const** *variables or static data members of integral or enumeration types initialized with constant expressions (8.5),* . . . 5.19p1

For conforming C programs this additional case does not cause a change of behavior. But if a C++ translator is being used to develop programs that are intended to be conforming C, there is the possibility that this construct will be used.

```
1  const int ten = 10;
2
3  char arr[ten]; /* constraint violation */
4                 // does not change the conformance status of the program
```

### Other Languages

Some languages regard enumeration and character constants as different types and would require a cast for such constants to appear as operands in an integer constant.

### Common Implementations

Some implementations map floating constant tokens to numeric values before casting them to an integer type.

### Coding Guidelines

Any reader comprehension issues apply whether or not an expression can be evaluated to a constant; for instance, use of parentheses. ?? expression shall be parenthesized

### Example

```
1  /*
2   * Both declarations define an array containing one element.
3   */
4  static int arr_1[2u + -1];
5  static int arr_2[1 + (int)0E99999999999999999999999999999999999999999];
```

---

1329 Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **sizeof** operator. integer constant cast of

### Commentary

Supporting casts to a floating type would require translators to be capable of handling floating-point arithmetic (unless the only subsequent operation on the result was a cast to an integer type). Although address constants are supported, the model used is one where they are handled at link-time. This means their value is not available, during the translation of a source file, to be cast to an integer constant. 1341 address constant

The evaluation of the **sizeof** operator involves deducing the type of its operand, which does not necessarily involve evaluating any of the contained subexpressions.

### Common Implementations

A few implementations (e.g., `gcc`) support casting constants to floating types.

---

1330 More latitude is permitted for constant expressions in initializers.

**Commentary**

program startup

The C language specification of initializers does not require translation-time knowledge of their values. The initialization can be evaluated as part of program's startup, allowing translators to treat initializers like any other expression. For instance, a translator need not concern itself with how many digits are significant in the evaluation of:

```
1   int x = (int)(1.0/3.0 + 2.0/3.0);
```

**C++**

5.19p2 *Other expressions are considered constant-expressions only for the purpose of non-local static object initialization (3.6.2).*

**Other Languages**

Most languages that support the use of initializers in declarations allow noninteger types to be assigned values having the appropriate type.

---

Such a constant expression shall be, or evaluate to, one of the following: 1331

**Commentary**

initializer type constraints

As well as evaluating to one of the constructs on this list, the type of the constant expression also has to be compatible with the type of the object it is initializing.

---

— an arithmetic constant expression, 1332

**Commentary**

arithmetic constant expression 1339

This case is discussed elsewhere.

---

constant null pointer constant

— a null pointer constant, 1333

**Commentary**

null pointer constant

The standard specifies two possible token sequences that can be used to represent the null pointer constant.

---

footnote 96

96) The operand of a **sizeof** operator is usually not evaluated (6.5.3.4). 1334

**Commentary**

sizeof result of
sizeof operand evaluated

The only time the operand of a **sizeof** operator may need to be evaluated is if its operand contains a VLA. If the operand of the **sizeof** operator contains a VLA, it is not a constant expression and must be evaluated at execution time.

**C90**

*The operand of a **sizeof** operator is not evaluated (6.3.3.4) and thus any operator in 6.3 may be used.*

Unless the operand contains a VLA, which is new in C99, it will still not be evaluated.

**C++**

The operand is never evaluated in C++. This difference was needed in C99 because of the introduction of variable length array types.

1335 97) An integer constant expression is used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **case** constant.

footnote 97

**Commentary**

This is a partial list of all places where the standard requires an integer constant expression (arrays having block scope no longer require their size to be specified using an integer constant expression).

**Other Languages**

Some other languages require translation-time constant expressions (in some cases constants) in these contexts, while others support execution-time values in these contexts (or their equivalent forms).

1336 Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

**Commentary**

These constraints are a result of the preprocessor having less information available to it and are discussed elsewhere.

conditional inclusion constant expression

1337 — an address constant, or

**Commentary**

This is discussed elsewhere.

1341 address constant

1338 — an address constant for an object type plus or minus an integer constant expression.

**Commentary**

An algorithm may call for the initialization value of the object being defined, with pointer type, to point to an element of an object other than its first. While recognizing the developer's desire for such functionality the Committee was aware that not all linkers were capable of providing it. They also recognized the commercial impracticality of requiring vendors to provide a linker to replace the one provided by the vendor of the host environment. It was believed that the majority of existing linkers supported a method of adding constant offsets to an existing address. This C sentence reflects this availability.

The requirement that the result of pointer arithmetic still point within (or one past the end) of the original object still apply.

pointer one past end of object

**C++**

The C++ language requires that vendors provide a linker for a variety of reasons; for instance, support for name mangling.

**Other Languages**

Some languages effectively support such initializers because they allow the address of an element of an array or member of a structure to be used as an address constant.

**Common Implementations**

An implementation's support for additional forms of address constant is likely to be dictated by the functionality available in the linker used.

**Example**

```
1  extern int glob;
2
3  static int *p_g = &glob + (sizeof(int) * 2);
```

arithmetic constant expression

An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **sizeof** expressions. 1339

**Commentary**

integer constant expression 1328

This defines the term *arithmetic constant expression*. An arithmetic constant expression is a more general kind of constant expression than an integer constant expression in that the restrictions on operands having a floating type have been lifted. Annex F, of the C Standard, also discusses arithmetic constant expressions.

Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a **sizeof** operator whose result is an integer constant. 1340

**Commentary**

integer constant cast of 1329

This specification is a more generalized form of the one given for integer constant expressions. It includes support for the conversion of arithmetic types to floating types.

address constant

An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; 1341

**Commentary**

This defines the term *address constant*. The term *scalar constant expression* would not be appropriate because arithmetic types are not included in the list. Common implementation practice (also for other languages) is for information on the storage address of objects having static storage duration and the address of the start of a function definition to be included in a program's image. This C definition of address constant reflects this existing implementation practice.

While objects having static storage duration are represented in a fixed, unique storage location, objects having other storage durations can be allocated storage at different locations every time their lifetime starts; the address of objects having automatic storage duration need not be unique (because of recursive invocations of the function that contains them).

The only context in which an address constant is required is the initializer for an object having a pointer type.

**C90**

The C90 Standard did not explicitly state that the null pointer was an address constant, although all known implementations treated it as such.

**C++**

The C++ Standard does not include (5.19p4) a null pointer in the list of possible address constant expressions. Although a null pointer value is listed as being a *constant-expression* (5.19p2). This difference in terminology does not appear to result in any differences.

**Other Languages**

Many languages only permit pointers to point at dynamically created objects. Such languages cannot contain address constants.

**Common Implementations**

program startup

In a hosted environment the relative addresses of objects, with static storage duration, and developer-written function definitions is usually decided by the linker. On program startup an area of storage is allocated to hold the objects having static storage duration. A fixed offset is added to the relative addresses of objects to obtain their actual addresses in the program's address space.

Library functions may be dynamically linked and their actual (possibly virtual) address is not known until they are called during program execution. In this case translators need to generate the machine code needed to ensure this execution-time address fix-up occurs.

In a freestanding environment the actual storage locations (addresses) assigned to objects (with static storage duration) and the machine code representing function definitions are usually known at link-time. For the other cases the complexity of program loading varies from always using a known fixed offset to the full virtual memory handling seen in hosted environments.

---

1342 it shall be created explicitly using the unary & operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type.

**Commentary**

These uses of the unary **&** operator are also techniques for generating addresses in nonconstant contexts. This list does not include using the value of an object to create an address (even if that object was assigned an address constant earlier in the same translation unit), or the value returned from a function call. Casts of integer constants to pointer types are often required in programs executing in freestanding environments, where it is known that certain addresses are mapped to hardware devices.

integer
permission to convert to pointer

**C90**

Support for creating an address constant by casting an integer constant to a pointer type is new in C99. However, many C90 implementations supported this usage. It was specified as a future change by the response to DR #145.

**C++**

Like C90, the C++ Standard does not specify support for an address constant being created by casting an integer constant to a pointer type.

**Other Languages**

Expressions having array or function type are not usually implicitly converted to addresses in other languages. Support for casting integer constants to addresses varies between languages.

**Common Implementations**

Many implementations support, as an extension, the casting of address constants to different pointer types as also being address constants.

**Example**

```
1  extern int glob;
2  int *g_p = (int *)1;
3  char *g_c = (char *)&glob;
```

---

1343 The array-subscript [] and member-access . and -> operators, the address & and indirection * unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

**Commentary**

The array-subscript **[]** and member-access **.** and **->** operators can all be used to specify particular subobjects within objects.

The operand used as the index in the array-subscript **[]** operator must be an integer constant expression if the resulting expression is to be an address constant.

The indirection * unary operator would normally be used to access the value of an object. However, when used in combination with the address **&** operator, the access can be *canceled out*.

&*

The member-access **->** operator requires a pointer type for its left operand. This will need to be an address constant if the resulting expression is to be an address constant.

Casts to structure type are not permitted in constant expressions, but casts of integer constants to pointer-to structures are permitted. A common implementation of the `offsetof` macro is:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

### Example

```
struct S {
           int m_1;
           short m_2[3];
         } x, y;

struct S *p1_s = &x;
int *p1_m1_s = &(y.m1);
int *p2_m1_s = &((&y)->m1);
int *p3_m1_s = &(((struct S *)0)->m1);
int *p4_m1_s = &(&*y.m1);

int arr[10];
int *p1_arr = arr;
int *p2_arr = &(arr[4]);

short *p3_m2_s = y.m_2;
short *p4_m2_s = &(y.m_2[0]);
```

---

constant expression other forms

An implementation may accept other forms of constant expressions. 1344

### Commentary

Here the standard gives explicit permission for translators to extend the kinds of expressions that are translation-time constants.

### C++

The C++ Standard does not given explicit latitude for an implementation to accept other forms of constant expression.

### Other Languages

Translators for other languages sometimes include extensions to what the language specification defines as a constant expression. Not many language definitions explicitly condone such extensions.

### Coding Guidelines

Developers may be making use of implementation-defined constant expressions when they use one of the standard macros (e.g., invoking the `offsetof` macro). In this case the choice is made by the implementation and is treated as being invisible to the developer (who may be able to make use of a translator option to save the output from the preprocessor to view the constant). The guideline recommendation dealing with the use of extensions is applicable here.

extensions ?? cost/benefit

### Example

```
static int foo = 2;
static int bar = foo + 2; /* A constant in the Foobar, Inc. implementation. */
```

---

constant expression semantic rules

conditional inclusion constant expression

The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.[98)] 1345

### Commentary

Additional constraints on constant expressions apply when they appear in conditional inclusion directives.

**C++**

The C++ Standard does not explicitly state this requirement, in its semantics rules, for the evaluation of a constant expression.

**Common Implementations**

Because the evaluation occurs during translation rather than during program execution, the effects of any undefined behaviors may be different. For instance, a translator may perform checks that enable a diagnostic to be generated, warning the developer of the conformance status of the construct.

---

1346 **Forward references:** array declarators (6.7.5.2), initialization (6.7.8).

---

1347 98) Thus, in the following initialization, footnote 98

```
static int i = 2  || 1 / 0;
```

the expression is a valid integer constant expression with value one.

**Commentary**

While this usage is unlikely to appear in the visible source, as such, it can occur indirectly as the result of macro replacement.

**C++**

The C++ Standard does not make this observation.

# References


1. K. D. Arbuthnott and J. I. D. Campbell. Effects of operand order and problem repetition on error priming in cognitive arithmetic. *Canadian Journal of Experimental Psychology*, 50(2):182–195, 1996.
2. M. H. Ashcraft. Cognitive arithmetic: A review of data and theory. *Cognition*, 44:75–106, 1992.
3. J. I. D. Campbell and Q. Xue. Cognitive arithmetic across cultures. *Journal of Experimental Psychology: General*, 130(2):299–315, 2001.
4. S. Dehaene. *The Number Sense*. Penguin, 1997.
5. M. Delazer and T. Benke. Arithmetic facts without meaning. *Cortex*, 33:697–710, 1997.
6. B. Edelman, H. Abdi, and D. Valentin. Multiplication number facts: Modeling human performance with connectionist networks. *Psychologica Belgica*, 36(1/2):31–63, Apr. 1996.
7. R. Ellis and G. Humphreys. *Connectionist Psychology*. Psychology Press, 1999.
8. W. S. Harley. *Associative Memory in Mental Arithmetic*. PhD thesis, Johns Hopkins University, Oct. 1991.
9. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
10. C. Lebiere. *The Dynamics of Cognition: An ACT-R Model of Cognitive Arithmetic*. PhD thesis, Carnegie Mellon University, Nov. 1998.
11. P. Lemaire and M. Fayol. When plausibility judgments supersede fact retrieval: The example of the odd-even effect on product verification. *Memory & Cognition*, 23(1):34–48, 1995.
12. M. Müller-Olm and O. Rüthing. On the complexity of constant propagation. In D. Sands, editor, $10^{th}$ *European Symposium on Programming (ESCOP 2001)*, pages 190–205. Springer-Verlag, Apr. 2001.
13. P. Suppes, M. Cohen, R. Laddaga, J. Anliker, and R. Floyd. A procedural theory of eye movements in doing arithmetic. *Journal of Mathematical Psychology*, 27:341–369, 1983.
14. R. L. Velduizen. C++ templates as partial evaluation. Technical Report TR519, Indiana University, July 2000.
15. C. Verbrugge, P. Co, and L. Hendren. Generalized constant propagation A study in C. In *Proceedings of the 1996 International Conference on Compiler Construction*, pages 74–89. Springer-Verlag, Apr. 1996.
16. J. W. Whalen. *The Influence of Semantic Number representations on Arithmetic Fact Retrieval*. PhD thesis, The John Hopkins University, July 2000.