Proposal for 1987 Smalltalk Standard syntax

Edited by:
L. Peter Deutsch, ParcPlace Systems

Contributors (alphabetical order, partial listing):
George Bosworth, Digitalk
Steve Burbeck, Softsmarts
L. Peter Deutsch, ParcPlace Systems
Barry Haynes, Apple Computer
Ralph Johnson, University of Illinois, Urbana-Champaign
J. Eliot B. Moss, University of Massachusetts, Amherst
Dave Thomas, Carleton University
David Ungar, Stanford University
Steve Vegdahl, Tektronix
Allen Wirfs-Brock, Tektronix

This is a working document proposing a complete definition of the syntax of the 1987 version of the Smalltalk-80 (TM) language. This language is a revision of the Smalltalk-80 language defined in the book "Smalltalk-80: The Language and Its Implementation" (Addison-Wesley, 1983), which we will refer to below as the Blue Book.

This document is meant to define the language syntax fully and formally, but only deals partially and informally with the semantics of the language. We intend to augment this document with a complete formal definition of the semantics of the language at some future time. We also recognize that the usefulness of the Smalltalk-80 language depends heavily on the set of defined classes and messages: in this respect it resembles Lisp, and contrasts with most other languages. We specify a very small set of classes and messages which we believe are required to support any useful Smalltalk-80 system, and which we consider a suitable starting point for standardization; however, we recognize that we have not dealt with this subject adequately.

Changes made in each version are not marked in the text, because they interfere too much with the reading of the equations and tables: the version history summarizes the important changes. Comments are solicited.

You will find this document easier to read if you print it in a fixed-pitch or nearly fixed-pitch font.

Version history:

[6] 1 July 1987: incorporates George Bosworth's cover letter, with some minor additions, and some minor clarifications discovered by Glenn Krasner; moves block arguments and assignment operators to lexical

syntax, to avoid problems with where whitespace can appear; changes primitives (again) to be specified either by class and selector, or by string. This version was sent to all workshop participants for approval.

[5] 5 June 1987: incorporates comments from the Feb. 26-27 workshop and subsequent contributions. Special thanks to Steve Vegdahl for debugging the previous version of the syntax by writing a parser for it.

Lexical syntax: clarified assumption that the scanner always takes the longest token in case of ambiguity; minor changes to number syntax, including removing alternate-radix floats and scientific notation for integers; explicitly reserved braces and backquote for future extensions; changed role of '-' in binary selectors.

Other syntax: removed dynamic array creation syntax, multiple expressions within parentheses; added #'...' to denote a symbol containing arbitrary characters; restored classes with both named and indexed instance variables, and changed the syntax of class definitions; specify primitives by string and number.

Semantics: removed all messages with fixed meanings; clarified position on redeclaration of names.

[4] 25 February 1987: made control messages have full message semantics; minor changes to number syntax; allow multiple expressions in parentheses, and clarify the role of '.'; provide for extended and contracted character sets; specify primitives by class and selector rather than number. This is the version presented at the Feb. 26-27 implementors workshop.

[3] 2 January 1987: moved Blue Book syntax to appendix; reorganized summary of changes; removed hook for declarations; removed declarations and multiple expressions within parentheses; made control messages less special; allow both numbers and strings for primitives; added section on file format; minor changes reflecting comments from internal review. This is the version sent to Eliot Moss for distribution to the participants in the Dec. 11-12 workshop.

[2] 22 December 1986: added summary comparison with Blue Book, class definition syntax.

[1] 20 December 1986: first version, no syntax for class definitions yet.

Syntax equations are given in BNF extended with the following constructs:

- [x] - optional x
- x* - 0 or more occurrences of x
- x+ - 1 or more occurrences of x

1. Summary of Changes from the Blue Book

This is a summary only: consult the following sections for details.

Clarifications

Some minor errors in classifying characters have been corrected.

The syntactic role of 'super' has been clarified.

The order of evaluation of receiver and arguments has been defined as left-to-right.

The syntax of primitives has been made explicit.

The syntax for defining classes has been made explicit.

Deletions

Alternate-radix floating point constants, and scientific notation for integers, are no longer provided.

Classes may be defined with instances containing named instance variables and/or indexed object references, or indexed 8-bit bytes only: they may no longer contain "words" (16-bit bytes).

Incompatibilities

Embedded doubled quotes may not appear within comments. (This turns out not to make any difference in what programs are accepted, only in how they are parsed.)

The syntax of literal arrays has been changed to make the individual elements look exactly like free-standing literals.

Block arguments and temporaries are properly scoped both lexically and dynamically, i.e. they are not stored in the home context. (Many existing programs will not execute properly, but they can always be changed so they will work under both current and proposed rules.)

The primitive, if any, comes before the method temporaries, not after. Standard primitives are identified if necessary by a class name and a selector; non-standard primitives are identified by a string.

Additions

We provide for extensions to the character set, and acknowledge the possibility of a reduced character set.

Lower-case letters are allowed within alternate-radix integers.

Symbol literals may contain arbitrary characters: the new syntax for this is # followed by a string.

Literal arrays may contain nil, true, and false in addition to other literals.

The sequence := now also means assignment, as an alternative to _ (left-arrow).

Blocks may have temporaries.

The control messages (ifTrue:, etc.) must not be forced to take explicit blocks as arguments. In fact, all messages, without exception, behave the same semantically, and may be redefined by the user.

Standard primitives may have a string description as well as a number.

2. Proposed 1987 standard

The following syntax is organized in the same way as the Blue Book syntax presented in the Appendix, with one major difference: it is specifically designed to be recognized by a recursive-descent parser with no backup and only a one-token buffer (such as the current Smalltalk-80 parser). Differences from the Blue Book are noted with **. Deleted constructs are marked with --; new ones with ++.

The syntax presented in the Blue Book does not indicate where separators (whitespace or comments) are allowed to appear. In the sections entitled Lexical Primitives below, separators are not allowed to appear between constructs; in the other sections, separators may appear between any two constructs (terminal or non-terminal symbols or groupings).

Character set

The 1987 Smalltalk standard syntax is based on the ASCII character set. Although this is not mentioned explicitly in the equations below, all non-printing ASCII characters (those with codes 0-31 and 127) are treated as whitespace-characters; all other characters are mentioned explicitly in the following section on lexical primitives.

We explicitly recognize that other character sets may include otherwise unspecified characters of three kinds: whitespace, alphabetic, and graphic. Characters classified as whitespace are ignored everywhere except in character and string constants; alphabetic characters are lexically equivalent to letters (i.e. may start and appear within identifiers); graphic characters are allowed in character and string literals and are illegal elsewhere. With regard to the ASCII set, all non-graphic characters with codes below 128 (i.e. codes 0-31 and 127) are whitespace; only letters (upper and lower case) are alphabetic. Any implementation that goes beyond the ASCII set (on which the syntax is based) may designate the additional characters as whitespace, alphabetic, and graphic in an implementation-dependent way. We do not define any particular mechanism for doing this.

We also recognize that the ISO character set differs from ASCII, and redefines certain ASCII characters to have different significance, particularly the square brackets and the braces. We do not specify alternative representations for these language elements at this time; however, we recognize that this must be addressed in the final version of this standard.

Lexical Primitives

We recognize that the lexical syntax is formally ambiguous, in that, for example, the string 'abc:' can be parsed either as an identifier followed by a non-quote-character, or as a keyword. We resolve this ambiguity in all cases in favor of the longest token that can be formed starting at a given point in the source text. Thus 'abc:' is always considered to be a keyword, if the 'a' is the beginning of the token.

The definition of token is not used anywhere else in the syntax: it is supplied only for exposition.

```
++  token = number | identifier | special-character | keyword |
          block-argument | assignment-operator |
          binary-selector | character-constant | string

    digit = '0' | ... | '9'
    digits = digit+
++  big-digits = (digit | letter)+  "as appropriate for radix"
**  number = (digits ['r' big-digits] |
          fraction-and-exponent) | fraction-and-exponent
++  fraction-and-exponent = '.' digits [('e' | 'E') ['-'] digits]
    letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
    identifier = letter (letter | digit)*
**  special-character = '+' | '/' | '\' | '*' | '~' | '<' | '>' |
          '=' | '@' | '%' | '|' | '&' | '?' | '!' | ','
--  (character)
++  non-quote-character =
          digit | letter | special-character |
          whitespace-character |
          '[' | ']' | '{' | '}' | '(' | ')' | '_' | '^' | ';' |
          '$' | '#' | ':' | '.' | '-' | '"'
++  block-argument = ':' identifier
++  assignment-operator = '_' | ':' '='
    keyword = identifier ':'
**  binary-selector =
          ('-' | special-character)
           ('-' special-character | special-character)*
**  character-constant = '$' (non-quote-character | '''' | '"')
    symbol = identifier | binary-selector | keyword+
**  string = '''' (non-quote-character | '''' '''' | '"')* ''''
**  comment = '"' (non-quote-character | '''')* '"'
    separators = (non-printing-character | comment)*
```

The syntax for numbers in the Blue Book, and its interpretation by the current Smalltalk compiler, leads to some unexpected results:

Input	Result of doIt	
1e3	1000	(e for exponent works with both integers and floats)
1.0e3	1000.0	
16r1e3	4096	(small e still means exponent)
16r10.0	16.0	(alternate-radix floats work)
16r1E3	483	(big E means hex digit)
.5	5	(initial . is interpreted as statement separator)

Under the new syntax, these examples would have the following interpretations:

1e3	(illegal - e is reserved for floats, and requires a .)

1.0e3 1000.0
16r1e3 483 (no upper/lower case distinction)
16r10.0 (illegal - no alternate-radix floats)
16r1E3 483
.5 0.5 (initial . means float)

As indicated in the syntax above, we propose to reserve e and E for floats, and to require a . for floats as well.

We have corrected the errors in the Blue Book, and removed the two-character limit on the length of binary-selectors. The character '-' still plays a special role: it cannot be allowed as the last character in a binary selector (unless it is the only character), because otherwise expressions like x+-5 would be ambiguous (x + -5 or x +- 5).

We allow ':=' for assignment, since the ASCII standard has underscore in place of left arrow (_). Unfortunately, this introduces a minor syntactic anomaly, explained in the section on expressions below.

Note that braces and backquote are deliberately reserved for future extensions to the language.

Atomic Terms

```
++  named-constant = 'nil' | 'true' | 'false'
**  symbol-constant = '#' (symbol | string)
--  (array)
**  array-constant = '#' '(' literal* ')'
**  literal = ['-'] number | named-constant | symbol-constant |
        character-constant | string | array-constant
    variable-name = identifier  "other than a named-constant,
                    pseudo-variable-name, or 'super'"
```

The new syntax for array constants is simpler to explain than the present Smalltalk-80 syntax, but different and a little more verbose. We explicitly recognize that nil, true, and false are constants. We add a new syntax (# followed by a string) for writing symbol constants containing arbitrary characters.

Expressions and Statements

```
**  primary = variable-name | pseudo-variable-name | literal |
        block-constructor | subexpression
++  pseudo-variable-name = 'self' | 'thisContext'
++  unary-message = unary-selector
    unary-selector = identifier
++  binary-message = binary-selector primary unary-message*
++  keyword-message = (keyword primary unary-message* binary-message*)+
--  (unary-object-description)
--  (binary-object-description)
--  (unary-expression)
--  (binary-expression)
--  (keyword-expression)
--  (message-expression)
```

```
--  (cascaded-message-expression)
++  cascaded-messages = (';' (unary-message | binary-message |
        keyword-message))*
++  messages =
        unary-message+ binary-message* [keyword-message] |
        binary-message+ [keyword-message] |
        keyword-message
++  rest-of-expression = [messages cascaded-messages]
**  expression =
        variable-name
          (assignment-operator expression | rest-of-expression) |
        keyword '=' expression "see below" |
        primary rest-of-expression |
        'super' messages cascaded-messages
++  expression-list = expression ('.' expression)* ['.']
**  temporaries = '|' temporary-list '|' | '||'
++  subexpression = '(' expression ')'
++  temporary-list = declared-variable-name*
++  declared-variable-name = variable-name
    statements = ['^' expression ['.'] | expression ['.' statements]]
**  block-constructor = '[' block-declarations statements ']'
++  block-declarations = temporaries |
        block-argument+
          ('|' [temporaries] | '||' temporary-list '|' | '|||')
```

In order to keep lexical analysis and parsing separate, but still allow constructs like x:=3, we have had to introduce the alternative

```
    keyword '=' expression
```

for assignment. This should really be read as though it were

```
    variable-name ':=' assignment
```

The syntax of messages has been rewritten to make it more amenable to parsing (and hopefully easier to read.) We explicitly recognize the existence of pseudo-variables, and the requirement that 'super' be followed by a message. (Note that cascaded messages to 'super' are allowed: some existing Smalltalk-80 compilers do not allow this.) The syntax for '.' allows it to be considered either a separator which may be followed by an empty element at the end of a list, or a terminator which may be optionally elided at the end of a list.

Blocks are allowed to declare local temporary variables: this is the one significant addition to the Blue Book syntax (and semantics). Note that a block with both arguments and temporaries requires a double '|' between the arguments and the temporaries. (Some syntactic circumlocutions are again needed to deal with the scanner's decision to accumulate consecutive '|'s into a single binary-selector.) We did consider the obvious alternative, which was to use only a single '|' in this case. The problem with this alternative is a potential ambiguity:

```
    [:x | t | y & z]
```

could be parsed as meaning either "argument x, temporary t, return y & z" or "argument x, return t | y & z." By requiring the double '|' for separating arguments from temporaries, we take the latter interpretation. (We also considered and rejected the three other ways to fix this:

- Disallow '|' as a binary operator character - rejected because '|' is the standard representation for "or".
- Remove the '|' after the argument list - rejected because it

reads worse.

- Make the distinction according to whether the names following the first '|' are already defined - rejected because this kind of syntactic dependency on far-away properties invites subtle problems.)

There is a semantic restriction, not expressible in a context-free syntax, that the only valid variable names are those declared within an enclosing scope (i.e. global, pool, class, or instance variables of the class where the method is being defined, or of some superclass; arguments or temporaries of the enclosing method, arguments or temporaries of an enclosing block, or temporaries of an enclosing subexpression). A full treatment of name scopes is beyond the boundaries of the present proposal. However, we do specify the following:

- A local name (method or block argument or temporary) must not conflict with a non-local name accessible in the same scope (global, pool, class, or instance variable).
- A local name may conflict with another local name accessible in the same scope: the inner declaration takes precedence.

We encourage compiler implementors to at least give a warning when compiling code that contains a redeclaration of a local name: this will help catch occurrences of the current Smalltalk practice in which a name used as a block argument is also declared in the method temporaries, e.g.

```
    | temp |
    ...
    ... [:temp | ...] ...
```

Methods

```
**  message-pattern =
        unary-selector |
        binary-selector declared-variable-name |
        (keyword declared-variable-name)+
++  primitive = '<' 'primitive:' [primitive-identification] '>'
++  primitive-identification = symbol symbol | string
**  method = message-pattern [primitive] [temporaries] statements
```

Primitives are identified either by a class and selector, or by a string. The former identify standard primitives; the interpretation of the latter is not defined. If the primitive-identification is missing, the class and selector name of the method in which the primitive appears are used. In general, we do not expect primitives to be standardized: instead, what we propose to standardize is the behavior of certain messages in certain classes, independent of whether they are implemented primitively or not.

Note that the implementation (compiler) is responsible for checking that primitives are only attached to methods and classes for which they are legal, i.e. this correspondence is truly part of the language definition. A particular implementation may include polymorphic primitives that accept (can validly be attached to) a variety of classes and methods.

We propose that the primitive specification precede, rather than follow,

the method temporaries. This seems more intuitive, since the primitive is executed before the temporaries are bound.

Classes

The Smalltalk-80 system takes quite a different approach to creating and editing classes vs. methods: the latter are defined by a textual syntax and a message interface for compiling it, while operations on the former are defined in terms of explicit messages taking various kinds of string arguments. Indeed, the Blue Book does not introduce any syntax per se for classes: it assumes they are created using the messages described in Chapter 16. However, the definition of classes is properly part of the language, just like the definition of methods, so we specify here the fundamental message for creating classes. The semantics of modifying existing classes, and the question of what happens to existing instances when a class is modified, are beyond the scope of the present proposal.

Note that we consider the creation of a class from a specification to be just like the creation of any other object, and unrelated to its installation under a name in any dictionary; indeed, we even consider the creation of a (compiled) method to be separate from its installation in a class.

A class is created by the following message:

```
Behavior
    newSuperclass: "Behavior | nil"
    instanceVariables: "Array of: Symbol"
    classVariables: "Array of: Symbol"
    poolDictionaries: "Array of: Symbol"
```

Giving a class a name, installing it in a dictionary, or classifying it in an organization are all matters outside the scope of the language definition. (We presume the existence of an interactive interface that allows users to define classes in some way that avoids writing out the above message, and also deals with naming and organization if relevant.)

The superclass specified for a class may be either another Behavior or nil. The latter is required for class Object, and allows creating other classes that are not subclasses of Object. (Doing this is fraught with peril, however: for example, if such a class does not define printOn:, the Smalltalk system is likely to go into a recursion loop the first time one tries to inspect an instance of the class.)

The syntax of the string supplied to describe the instance variables is

```
inst-var-names =
    declared-variable-name* [indexed-refs] |
    indexed-bytes
indexed-refs = '*' 'Object'
indexed-bytes = '*' 'Byte'
```

Thus a class may contain named instance variables that hold object references, indexed instance variables that hold object references (e.g. Array), both (e.g. OrderedCollection), or information that is not object references (e.g. ByteArray). We anticipate that an implementation will provide a variety of different kinds of primitive access to bit-type objects, e.g. by 8-, 16- or 32-bit bytes, or perhaps to arbitrary bit sequences: the only reason for calling such objects

'byte' as opposed to 'bit' objects is that we do not require implementations to quantize the space for such objects in units smaller than 8 bits.

File syntax

The form in which Smalltalk programs are stored on external files is defined, not in the Blue Book, but in chapter 3 (pp. 29-37) of the Green Book. We propose to standardize enough of this external format so that external files containing programs can be parsed even by systems that may not be able to interpret all of their contents. In the syntax equations below, separators are NOT implicitly allowed between elements: the equations must be taken exactly as they appear.

```
marker = '!'
non-marker = "any character except the marker"
separators = non-printing-character*
chunk = (non-marker | marker marker)+ marker
special-read-section = marker chunk (separators chunk)*
    separators marker
program-file = (separators (special-read-section | chunk))*
    separators
```

Information appears on a program file in "chunks" terminated by a marker and with embedded markers doubled. A chunk not preceded by a marker is simply an expression to be evaluated. A chunk preceded by a marker indicates the start of special syntax: the expression is evaluated to produce some kind of reader or parser object, which in turn is sent the message scanFrom: with the file stream itself as the argument. The reader then is expected to read and process chunks from the file until encountering an empty chunk. In other words, the following might represent the algorithm for reading in a program file:

```
[self skipSeparators.
 self atEnd]
   whileFalse:
     [(self peekFor: $!)
        ifTrue: [(Object evaluate: self nextChunk) scanFrom: self]
        ifFalse: [(Object evaluate: self nextChunk)]]
```

The purpose of the special-read-section is primarily to allow classes to read in method definitions without having to have them copied two extra times (once for chunk parsing, once for parsing as a string literal to be passed as an argument.) The current Smalltalk-80 system copies the definition one extra time, since it reads it in as a chunk before parsing: this can clearly be avoided if desired.

At a minimum, a file parser must be able to identify method definitions. We propose to do this in the following way: we define the message <Behavior> methodReader as returning an object whose scanFrom: method will read and define methods for the receiver. We further specify that additional messages may be sent to this object without compromising its function, e.g.

```
!aBehavior methodReader category: 'something'!
```

By this convention, an implementation can define additional properties

for methods being read without compromising general parsability of source files.

Semantics

Order of Evaluation

The expressions in an expression-list are evaluated in left-to-right order.

The message sends in a cascade are evaluated in left-to-right order.

The receiver of a message is evaluated before the arguments; the arguments are evaluated in left-to-right order.

Standard Classes

Certain classes are conceptually required to support the language defined above, namely, the classes of literal objects. These classes are:

- Integer
- Float
- Symbol
- String
- Array
- Character
- Block
- True, False, Nil
- Behavior (for classes)

We do not require that these classes have these specific names, or that their functionality is divided up in exactly this manner (for example, integers might be implemented by separate SmallInteger and LargeInteger classes, or True and False might be instances of a single class Boolean). However, in describing the standard messages in the next section, we will use these names and this division of functionality.

Standard Messages

The language as we have described it has no messages with fixed meanings. We regard this as a unique strength of Smalltalk (and of related languages such as Hewitt's Actor languages), and experience has indicated the utility of this concept in enabling such things as transparent message forwarders. On the other hand, any useful language must provide a basic set of functions such as arithmetic and control structures, and any commercially viable language must implement some of these functions very efficiently. We now define a small set of messages that we expect all implementations of the Smalltalk-80 language to provide. In the next section, we discuss how the pragmatics of these or other messages may be modified to enable efficient implementation.

Here are the messages we believe are appropriate to standardize as an

absolute minimum for language support. The marks in the left margin refer to the section on pragmatics and should be ignored at this point.

```
    Arithmetic:
P       (Integer) + - * < > <= >= = ~= (Integer, Float)
P       (Integer) // \\ (Integer)
P       (Integer) / (Float)
P       (Float) + - * / < > <= >= = ~= (Integer, Float)
    Control:
P       (Block) value
P       (Block) value: (Object)
 F      (Block) whileTrue: (Block)
 F      (Block) whileFalse: (Block)
 F      (Block) whileTrue
 F      (Block) whileFalse
 F      (Block) repeat
P       (Integer) to: (Integer) do: (Block)
        (Integer) timesRepeat: (Block)
 F      (True, False) ifTrue: (Block)
 F      (True, False) ifTrue: (Block) ifFalse: (Block)
 F      (True, False) ifFalse: (Block)
 F      (True, False) ifFalse: (Block) ifTrue: (Block)
 F      (True, False) and: (Block)
 F      (True, False) or: (Block)
    Miscellaneous:
 F      (Object) == (Object)
```

Contexts

Contexts are the one area in which we propose several minor changes in the semantics of Smalltalk, all of which are backward-compatible given some minor changes in the Virtual Image.

The first change has to do with the scope and lifetime of block arguments (and temporaries, which are new). In the current Smalltalk-80 definition, block arguments are stored in the home context. This prevents blocks from being used recursively, or by more than one process, and leads to anomalous error messages if a process is interrupted ("Block already active"). In the proposed definition, blocks are closures in the sense of Scheme or other modern lexically scoped Lisps: execution of the [] construct creates a BlockClosure, which only encapsulates the current (home) context and the code; invocation of a BlockClosure creates a BlockContext. We note that an implementation may optimize this process, as long as the semantics are maintained, in ways familiar from the Lisp literature: for example, a block that refers to no variables in outer scopes, and does not do a ^ return, may not need to hold a reference to the outer scope. One result of this is that the debugger may have less information available to it: we explicitly allow this to be the case.

The second semantic change has to do with ^. When control returns from a method, but the context being returned to is anomalous (e.g. has already been returned from), the current system sends the message 'cannotReturn: theValue' to the context being returned from. We propose to change this so that the system sends the message 'resumeWith: theValue' to the object (presumably, but not necessarily, a context)

being returned TO. For convenience in implementing non-standard control structures, this message should be defined primitively in class Context.

The third change also affects ^. Currently, ^ from within a block invokes a complex algorithm that users have no control over. We propose to change this so that ^ within a block is defined as sending the message 'thisContext remoteReturn: theValue'. Normally this message will be defined in class BlockContext as a primitive that carries out to the current built-in algorithm, but future evolution of the system to incorporate exception handling with unwind-protection might affect the definition. In this regard, we allow (but do not require) implementations to disallow ^-returns to contexts whose sender chain terminates elsewhere than the root of the current process: situations of this kind should be handled with explicit use of resumeWith:.

As indicated below, compilers may be able to avoid creating blocks altogether under certain circumstances, such as in the standard conditional message ifTrue:ifFalse:. As a consequence, however, the value of thisContext would be an outer context rather than the actual (textual) current context. We propose to require absolutely faithful implementation, which may require constructing an actual context for a conditional message if a thisContext appears within one of the alternatives.

Pragmatics
==========

While we require absolutely uniform message semantics for the Smalltalk-80 language, we recognize the need to allow more efficient implementation of messages whose meaning is very unlikely to change. To this end, we propose that certain messages, while retaining the same semantics as all others, are allowed to have substantially different pragmatics. In particular:

- Certain messages, if sent to receivers whose classes are not in a specified set, may execute much slower.
- Certain messages, if defined in new classes, or redefined or undefined in existing classes, may suffer a substantial performance penalty for some or all classes of receiver. In exchange for these penalties, under normal circumstances these messages will execute substantially faster than others.

In the list of standard messages defined above, the messages marked "P" have the first pragmatic property ("P" indicating that the set of classes for that message is Partially fixed); the messages marked "F" have both the first and the second property ("F" indicating that the set of classes for that message must stay Fully fixed to avoid losing performance).

"Changing a definition" means that the new definition is not operationally equivalent to the old. A sufficient (but not necessary) condition for testing this is that the new definition compiles into the same object code as the old one: we encourage implementors to use a test such as this one, so that (for example) changing variable names or comments is not considered "changing the definition". The pragmatic consequences of (for example) not compiling ifTrue:ifFalse: in-line are so severe that the user should be given an opportunity to confirm that

this was actually the intended result. (This is a user interface question, not a matter of language definition.) We note that no existing Smalltalk-80 compiler known to us handles this possibility properly.

We also note that the 'notBoolean' exception, which results from non-Boolean conditions in the present Smalltalk-80 definition, does not conform to the proposed standard: if an implementation uses a mechanism like a notBoolean message internally, it must convert this automatically to a correct ifTrue:ifFalse: (or whatever) message, with two appropriate blocks as arguments, without user intervention.

Current Smalltalk-80 compilers that adopt the Blue Book's concept of "special arithmetic selectors" do not properly handle redefinition of these messages: this does not conform to the proposed standard.

Nothing in this standard precludes a given implementation from adding or removing messages or receiver classes to the lists given above. Such differences may be built into the implementation, or may be under user control with a sufficiently sophisticated compiler. However, since such enhancements are required to leave the semantics unchanged, we do not specify anything about them here.

Static checking

Since supplying receivers or arguments of the wrong class to the abovementioned messages will almost certainly result in a runtime error, compilers may choose to issue warnings if they believe the user has written a program that is likely to result in an error. For example, a programmer unused to Smalltalk syntax might write something like

```
    a < b ifTrue: trueStuff ifFalse: falseStuff
```

rather than

```
    a < b ifTrue: [trueStuff] ifFalse: [falseStuff]
```

A compiler might plausibly ask for user confirmation if an argument to ifTrue:ifFalse: is not an explicitly written block. However, the proposed standard requires that all compilers must be willing to compile programs even if they contain questionable constructs of this kind. Most current Smalltalk-80 compilers do not do this.

Contexts

All existing compilers for the Smalltalk-80 language treat some or all of the control messages specially by compiling them in a way that avoids creating Context objects for them during execution. Since the language standard does not specify anything about the creation of contexts, a compiler is free to compile ANY message in a way that avoids creating Contexts, so long as the semantics of the message are unaffected. We note, however, that since Contexts are visible to the programmer at the meta-level, programs at the meta-level must be prepared for the possibility that a given message send at the source level may not create a Context at the object level.

Appendix: The Blue Book
==========

The following syntax is the one that appears in the endpaper of the Blue Book (slightly rearranged). A few notes on errors and omissions are interspersed. This material is reprinted by permission of Xerox Corporation.

Lexical Primitives

```
digit = '0' | ... | '9'
digits = digit+
number = [digits 'r'] ['-'] digits ['.' digits] ['e' ['-'] digits]
letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
identifier = letter (letter | digit)*
special-character = '+' | '/' | '\' | '*' | '~' | '<' | '>' |
    '=' | '@' | '%' | '|' | '&' | '?' | '!'
character =
    digit | letter | special-character |
    '[' | ']' | '{' | '}' | '(' | ')' | '_' | '^' | ',' | ';' |
    '$' | '!' | '#' | ':'
keyword = identifier ':'
unary-selector = identifier
binary-selector = '-' | special-character [special-character]
character-constant = '$' (character | '"' | '''')
symbol = identifier | binary-selector | keyword+
string = '"' (character | '"' '"' | '''')* '"'
comment = '"' (character | '"' '"' | '''' '"')* '"'
separators = (non-printing-character | comment)*
```

The syntax for special-character and character have several errors. '!' appears in both (it should only be in special-character); ',' appears in character (it should be in special-character); '.' appears in neither (it should be in character); '-' and ''' (backquote) appear in neither (they should be in special-character).

There does not appear to be a good reason for limiting the length of binary-selectors to two characters. '-' is apparently singled out because of its special role in indicating negative numbers.

Atomic Terms

```
symbol-constant = '#' symbol
array = '(' (number | symbol | string | character-constant |
    array)* ')'
array-constant = '#' array
literal = number | symbol-constant | character-constant |
    string | array-constant
variable-name = identifier
```

Expressions and Statements

```
primary = variable-name | literal | block | '(' expression ')'
unary-object-description = primary | unary-expression
binary-object-description = unary-object-description |
```

```
        binary-expression
    unary-expression = unary-object-description unary-selector
    binary-expression = binary-object-description binary-selector
        unary-object-description
    keyword-expression = binary-object-description
        (keyword binary-object-description)+
    message-expression = unary-expression | binary-expression |
        keyword-expression
    cascaded-message-expression = message-expression (';'
        (unary-selector | binary-selector unary-object-description |
         (keyword binary-object-description)+ )+
    expression = (variable-name '_')*
        (primary | message-expression | cascaded-message-expression)
    statements = ['^' expression | expression ['.' statements]]
    block = '[' [(':' variable-name)+ '|' statements | statements] ']'
```

Methods

```
    temporaries = '|' variable-name* '|'
    message-pattern = unary-selector | binary-selector variable-name |
        (keyword variable-name)+
    method = message-pattern [temporaries [statements] | statements]
```

The Blue Book omits the syntax for indicating primitive methods. (This may be deliberate.)