X3J20/93-004

Announcement of Proposed 1987 Smalltalk Language Specification

by (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

July 1, 1987

The attached document is a proposed specification for the Smalltalk language. It is the outgrowth of a series of meetings held at Stanford University on December 11 and 12, 1986 and February 26 and 27, 1987. The need for such a specification is a direct result of Smalltalk's success. The language, environment, and concepts that were given birth at Xerox PARC over a decade ago have grown beyond their initial research setting into various implementations and systems commercially available from numerous vendors.

Our immediate goal in formulating the attached document is to provide a basis for greater portability of applications among Smalltalk implementations by means of a clear and precise language specification. At present, the Smalltalk language is under specified leading to divergence among implementations, thus needlessly complicating portability.

We recognize that we represent but a fraction of the total Smalltalk community and have therefore purposely limited ourselves to the following goals:

1. Clarify, cleanup, and unify the details of the existing Smalltalk language definitions;

2. Minimize the disruptions to existing users with large bodies of working Smalltalk code;

3. Add a few mature and valuable enhancements.

We believe we have modestly met the above goals. We intend that our future implementations will follow this proposal and we encourage others to do the same. We, therefore, earnestly solicit comments on the attached proposal. Comments should be forwarded to:

L. Peter Deutsch
ParcPlace Systems
P. O. box 60264
Palo Alto, CA 94306

Further, we feel that the attached proposal represents but a necessary first step in achieving an eventual Smalltalk standard. Clearly much more remains to be done, but by formulating a limited but concrete proposal at this time, we hope to lay the groundwork upon which further efforts can be based. More importantly the process of preparing this proposal has already fostered the sense of good will and willingness to proceed among the major Smalltalk system vendors that is crucial for the eventual success of a future standard.

We wish to see the process continued, especially since we see that Smalltalk systems are gaining wide commercial acceptance. We intend to continue to work together among ourselves, with users, and the wider community to evolve the Smalltalk language and Smalltalk systems. In this connection, we note that IEEE has approved and is in the process of organizing an official Smalltalk Standard effort. We hope that all interested parties will participate in this effort: details will be announced in the near future. The present draft language standard is not meant to preempt the IEEE activity, but rather to serve as an input to it, and as a possible interim partial standard while the more careful IEEE process is underway.

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.)

Protected applications in the Smalltalk-80 System

L. Peter Deutsch
Xerox, ParcPlace Systems

This is a working document proposing mechanisms for better encapsulation and protection of independent applications for the Smalltalk-80 language and system.

Comments are solicited. For changes in successive versions, consult the version history.

Version history:

[2] 10 May 1987: split into three documents (scopes, pkg, and load); incorporated comments by Richard Steiger, Mark Miller, and George Bosworth; decoupled this document from the standards activity; added new section on debugging and errors; added syntax for creating private security area and name scope.

[1] 25 February 1987: first version (pkg87), distributed to implementors at the Feb. 26-27 workshop.

Introduction

Conventional language systems support the development of applications that can be made available in object code form only, and are thereby protected from unwanted interference or scrutiny (subject to the ability of the programmer to create pointers to arbitrary machine locations). In contrast, the current Smalltalk-80 system is completely open: all objects (including source code) are available for inspection and modification, and anyone can send any message to any object to which the sender has a reference. This greatly reduces the incentive to design in a strongly modular way, and tends to lead to a tangled, monolithic system. In order to alleviate this problem, and also to encourage the development of third-party software, this document proposes a mechanism for truly encapsulated applications within the Smalltalk-80 system.

In theory, an object-oriented language such as Smalltalk provides excellent abstraction, since messages are interpreted relative to the class, and inter-program protection, since there is no way to violate the abstraction boundary of an object. In practice, there are several barriers to meeting this promise:

- All class names are global, so any object can send a message to any class. This makes protection awkward, because typical applications have both public and private classes.
- Any object can send any message to any other object it has a reference to. This makes protection awkward, because typically objects

have both public and private protocols.
- Debuggers need a controlled way of breaching the abstraction boundary, but in the current system there is no way to restrict access to messages such as instVarAt:.

We propose to address these problems by a combination of language and system facilities. A separate document describes new mechanisms for more flexible control of name visibility, which addresses the first problem mentioned above. Another document (in preparation) describes a proposal, largely due to Mark Miller, for a systematic approach to protecting objects from direct inspection other than by authorized subsystems such as debuggers. Consequently, this document only deals with the second problem: how to provide encapsulation without having to add new mechanisms to restrict the ability of objects to send arbitrary messages to arbitrary other objects.

Our solution to the encapsulation problem is based on separating public and private protocols into different classes. This approach has significant overhead, and we believe it is most appropriate for applications with stringent information hiding needs. We believe that a superior solution would be to allow message selectors to be more general objects, not necessarily literal Symbols, which would allow a package to have selectors that cannot be named by its clients or anyone else. However, this approach appears to require more new machinery than the one we propose here, as well as being less flexible in some ways (e.g. it doesn't directly provide for dynamic revocation of capability).

The proposed approach involves no changes to any existing part of the Smalltalk language or system; however, it depends on the facilities for flexible name scoping and meta-level protection.

Meta-level facilities

Mark Miller's proposal includes an explicit notion of a "security area", an object that controls access to the representation of other objects. We propose that this notion be implemented in a way that allows private application classes and their instances to belong to security areas different from the normal "open" area, and to different areas for different applications. We only sketch some of the ideas here: for more information on this concept, read Mark's proposal.

For private messages to have any value as a protection mechanism, they must be protected in the same way as an object's implementation. (This is, for example, consistent with the Actors view that objects are individually fully responsible for decoding their messages.) For this reason, the operations that allow direct access to an object's class (including the method dictionary) are also placed in the "meta" category.

Since Mark's proposals are complex and far-reaching, we only discuss a subset of them here. This subset is aimed specifically at the following problems:
- Controlling access to an object's storage representation (the functionality of instVarAt:, instVarAt:put:, become:, class).
- Controlling access to the state of a class (method

dictionary, class variables, superclass reference).

As in Mark's proposal, we introduce the concept of a MetaObject. A MetaObject holds a reference to an object (called the 'subject' of the MetaObject), and provides direct access to the instance variables of the subject. Access is controlled by controlling the creation of MetaObjects for a given subject. Rather than provide instVarAt: and instVarAt:put:, we provide a message

<InstVarDictionary> := <MetaObject> instVarDictionary

that returns an object that provides Dictionary protocol to actually access the instance variables of the MetaObject's subject by name. (We do not specify how this occurs: presumably the InstVarDictionary invokes protected messages of the MetaObject, that in turn call primitives similar to the present instVarAt:[put].)

To control the creation of MetaObjects, we introduce the notion of a SecurityArea, again as in Mark's proposal. Only a SecurityArea can create a MetaObject. The Smalltalk system starts out with a single SecurityArea, which can create a MetaObject for any existing object. Ideally, the primitives for creating objects should specify in what SecurityArea the object is to be created. However, since we have had no experience with the SecurityArea concept, we are unwilling to require implementors to provide this capability, which may add significant complexity to the memory manager. Instead, we adopt a much more limited position that addresses some, but not all, the security needs of protected applications: we divide objects among SecurityAreas on a per-class basis, and propose that each class specify which SecurityArea can access it and its instances, i.e.

<SecurityArea> := <Behavior> securityArea.

Protected interfaces

As mentioned above, and explained in more detail below, applications can effectively control what objects are available to their clients by name. In this section, we describe how they can use this facility to provide a completely protected interface to clients. No new language or system facilities are involved: this is purely a matter of convention. At the end of this section, we provide a detailed example.

If an application wishes to present a protected interface, it is not sufficient to only make a particular class or classes visible:

- The client can instantiate the class in an uncontrolled way.
- The client can send any message to instances of the class, not just those intended for public use.
- The client can do other operations on the class, such as changing its methods.

To prevent clients from instantiating an application class, we propose that the application not make classes available to the client. Instead, the application should export a "factory" object that only understands a few object-creation messages. This factory, having been compiled in the naming environment of the application, can refer to and instantiate

application classes as needed. As we will see below, factories are sufficiently stylized that they can be generated automatically from declarations of what messages should be public.

To prevent clients from sending private messages, we propose that the objects provided dynamically by the application to the client not be instances of the application classes that actually do the work. Instead, the application should provide interface objects that only understand the public messages. The interface object should hold, in an instance variable, a reference to an object that actually provides the functionality, and forward the public messages to it. Interface objects obviously need an initialization message to set this reference, and this message must be protected in some way so that clients cannot change the reference once it has been set. For this message, we propose a different technique, based on checking a "key" against an object not visible to clients: see the example below for details. Again, most of the code for interface classes can be generated automatically from declarations.

It has been suggested we do not actually need to use a key for validating the initialization message for the interface object: we can simply check whether the instance has been initialized yet (i.e. the instance variable holding the real object is non-nil), and signal an error if this is the case. This approach seems to work for the simple case presented below, but may not be adequate for more complex situations.

As discussed in the section on meta-level facilities above, by placing an application in a different security area, we can prevent clients from accessing its machinery (such as the method dictionaries of its classes, or the instance variables of its instances) in an uncontrolled way.

Here we give a very simple example of a protected application: a Counter that is initially zero, and can only be incremented and read. Even in this simple example, an interface class is required, because the initialization message to the Counter must be protected. None of the three classes exhibited below are visible to clients: only the static variable named CounterMaker is exported. The static variables (classes) CounterFactory, CounterInterface, and Counter exist only in a scope private to the application, as does the variable CounterKey.

The code given here for creating security areas and name scopes is completely speculative, and may not be at all appropriate in general. Indeed, the syntax is not quite acceptable, since it includes temporary variables declared in the middle of a statement sequence, and a mixture of language syntax and file "chunk" delimiters.

```
" ------ Create the SecurityArea ------ "

| myArea |
myArea := SecurityArea current newArea.
myArea control: [

" ------ Create the private name scope ------ "

| myGlobals |
```

```
myGlobals := Dictionary new.
globals at: #CounterArea put: area.
globals at: #CounterGlobals put: globals.
globals enclose: '

" ****** ALL THE 'S SHOULD BE DOUBLED FROM HERE ON ****** "

" ------ The factory class ------ "

CounterFactory := Behavior
    newSuperclass: Object
    instanceVariables: #()
    classVariables: #()
    poolDictionaries: #().

!CounterFactory methods forCategory: 'Counter creation'!
new
    "Make a new Counter for the client"
    ^CounterInterface new initialize: CounterKey! !

" ------ The interface class ------ "

CounterInterface := Behavior
    newSuperclass: Object
    instanceVariables: #(#counter)
    classVariables: #()
    poolDictionaries: #().

!CounterInterface methods forCategory: 'Counter creation'!
initialize: aKey
    CounterKey == aKey
        ifFalse: [self error: 'Unauthorized message'].
    counter := Counter new.
    counter initialize.
    ^self! !

!CounterInterface methods forCategory: 'client accessing'!
increment
    counter increment!
value
    ^counter value! !

" ------ The real counter class ------ "

Counter := Behavior
    newSuperclass: Object
    instanceVariables: #(#count)
    classVariables: #()
    poolDictionaries: #().

!Counter methods forCategory: 'accessing'!
initialize
    count := 0!
increment
    count := count + 1!
value
```

```
        ^count! !

    "Create the key.  This can be any mutable object whatever.  It
    can't be an immutable object, since we can't be guaranteed that
    two immutable objects with the same contents won't be ==."

    CounterKey := Array with #Counter "a mutable array"!

    "Create the CounterMaker.  CounterMaker is the only static
    variable defined here that is visible to clients."

    CounterMaker := CounterFactory new!

    Smalltalk at: #CounterMaker put: CounterMaker!

    " ------ End of scope of CounterGlobals ------ "

        '.

    " ------ End of scope of CounterArea ------ "

        ]!
```

Debugging and errors

If an error occurs inside a protected application, we must ensure that the debugger does not allow us to violate a protection boundary. For example, Contexts probably need to be created in the SecurityArea of their receiver. The debugger probably should not even show Contexts in SecurityAreas to which the user does not have access. A complete discussion of this problem is beyond the scope of this document.

Proposal for 1987 Smalltalk Standard primitives

prepared by:
L. Peter Deutsch, Xerox, ParcPlace Systems

This is a working document proposing the complete set of standard required primitives, and some standard optional primitives, for the 1987 version of the Smalltalk-80 language. Comments are solicited. Changes made in successive versions are not indicated in the text: consult the version history for a summary.

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

Version history:

[4] 25 February 1987: removed all primitives except those needed to support the language definition. This is the version distributed at the Feb. 26-27 implementors workshop

[3] 4 January 1987: changed standard primitives back to numbers; distinguished bitmaps from byte arrays. This is the version sent to Eliot Moss for distribution.

[2] 22 December 1986: added space inquiries; finished specifying the graphics primitives.

[1] 21 December 1986: first version.

The set of primitive methods is clearly one of the most implementation-dependent parts of Smalltalk-80 systems. Since our purpose in setting standards is to specify those things that all implementors can agree to support, we believe it is appropriate only to specify those primitive methods necessary to support the Smalltalk-80 language. In particular, we do not specify any primitives relating to I/O. We hope that a future effort will result in specifying standards in this area.

We recognize that implementors may choose to provide two different kinds of primitives in their systems:

- Primitives whose function is documented, but whose implementation is not accessible to users. We propose to specify these in terms of a standard class and selector, e.g. the primitive for adding SmallIntegers is called SmallInteger +. Some primitives of this kind are standardized here; implementors may add others at will. Compilers should give at least warning messages for primitives they do not recognize, and compile such methods as though the primitive specification were not there.
- Primitives defined by users, using an implementation-dependent

syntax. We propose to reserve a syntax for expressing these in the form of a string, but do not specify their semantics. For example, an implementation might allow users to write methods like

```
(SmallInteger) bitInvert
    <primitive: 'top ^= ~3;'>
    ^-1 - self
```

or

```
(SmallInteger) bitInvert
    <primitive: 'eorl #~3,top'>
    ^-1 - self
```

When porting an image or source code from one implementation to another, the default action should be to ignore primitives defined by strings.

For the remainder of this document, where we say "primitive" we mean "standard primitive", unless we explicitly say otherwise.

Even though the argument classes for primitives are often mentioned in the Blue Book, and are easily determined for those primitives for which the Blue Book supplies implementations, they are not documented in any one place, and there have been subtle disagreements between implementations, particularly on the issue of allowable receivers and arguments for the integer primitives. Consequently, we take this opportunity to specify the supported argument classes for all primitives. We also require the following properties of all (standard) primitives (and encourage, but do not require, implementors to observe them for implementation-specific primitives, as well as for standard ones):

- At the implementor's discretion, any primitive which is required to accept SmallIntegers as arguments is also allowed to accept LargeIntegers (positive, negative, or both, and perhaps only up to a certain size), and vice versa.
- If an integer result is expressible as a SmallInteger, it must be so expressed, even if it results from operations on LargeIntegers.
- At the implementor's discretion, any primitive which returns an integer result has the option of either only handling SmallInteger results (failing if the result cannot be represented as a SmallInteger), or of returning an appropriate LargeInteger in any particular case.
- At the implementor's discretion, any primitive which is required to accept Floats as arguments is also allowed to accept any subset of the Integer classes (small and/or large).

The current Smalltalk-80 VM does not provide any way for an implementor to check that an argument is a Semaphore, because Semaphore is not among the "well-known" classes listed on p. 576 of the Blue Book. This has caused considerable trouble in at least one implementation. We propose that ByteArray, Semaphore, and LargeNegativeInteger be recognized as classes in which a VM implementation may be interested, and allow implementors to take whatever steps are appropriate to make this possible.

We note that the implementor is NOT allowed discretion in the choice of acceptable receivers for a primitive, only in whether or not to allow a broader choice of arguments. In particular, we require that the compiler only allow a primitive to be attached to a method if the class in which the method is being compiled is among the acceptable receiver classes for that primitive: this may require a table lookup. We also

require that if a primitive accepts a given class as its receiver, it also accept all subclasses of that class.

Old primitives

Here is a complete list of the Blue Book primitives that we propose to retain as standard. For each primitive or range of primitives, we indicate its acceptable receiver class(es) (implicitly including all subclasses as well) and its acceptable argument class(es). Please refer to chapter 29 of the Blue Book, and especially the list of primitives on pp. 612-615, for additional detail on the list below.

Replaced

60-62 (Collection at:, at:put:, size)
These have been replaced by new, non-polymorphic primitives that are specific to the kind of collection being accessed: see below.

73, 74 (Object instVarAt:, instVarAt:put:)
A higher-level protocol, defined in the companion document on execution protocol, replaces these primitives.

75 (Object asOop)
This has been replaced by a primitive that returns a not necessarily unique hash value: see below.

87, 88 (Process resume, suspend)
See the companion document on execution protocol for the revisions of the process machinery.

Clarified

1-17 (SmallInteger arithmetic and comparison)
Receiver: SmallInteger; Argument: SmallInteger. As discussed above, these primitives may accept LargeIntegers as arguments, but they do NOT accept LargeIntegers as receiver: that is what primitives 21-37 do.

Changed

81 (BlockClosure value, value:*)
Receiver: BlockClosure, not BlockContext. See the companion documents on language definition and execution protocol for more information.

82 (BlockClosure valueWithArguments:)
Receiver: BlockClosure; Argument: Array. The argument must be an Array, not any other kind of collection.

Unchanged

40 (SmallInteger asFloat)
41-54 (Float arithmetic and comparison)
63-64 (String at:, at:put:)
70-71 (Behavior new, new:)
85-86 (Semaphore signal, wait)
110 (Object ==)
111 (Object class)

New primitives

Numbers

We propose to add the following new standard primitives for integers:
SmallInteger rem: SmallInteger -- there is no good reason to have //, \\, and quo: but not rem:.

Collections

We propose to change the primitives related to variable-size objects so that each one works only on a particular kind of object. Since we propose to eliminate objects containing both named and indexed instance variables from the language (see 'Classes' above), we propose that the primitives in the following list be installed locally in the class hierarchy where needed, and definitely not in class Object. The classes needing these primitives are probably:
ByteArray/Large(Positive/Negative)Integer/Bitmap for the byte-containing group, and Array for the pointer-containing group.

Array at: SmallInteger
Array at: SmallInteger put: Object
Array size
ByteArray at: SmallInteger
ByteArray at: SmallInteger put: SmallInteger
ByteArray size

The Smalltalk-80 system implicitly assumes that Strings and ByteArrays use the same stored representation. We hereby make this assumption explicit, at least to the extent of asserting that each element of a String occupies a single byte. However, to complete the semantic distinction between the two, we propose to add one new primitive for operating on Strings, namely:

String size

Miscellaneous

Object hash

This operation has the following properties:

- The primitive returns a non-negative SmallInteger.
- Distinct objects may have equal (primitive) hash values.
- An object's (primitive) hash value never changes.
- The (primitive) hash values of equal SmallIntegers are the same.

For the correct functioning of the Smalltalk system, classes that redefine = must also redefine hash so that any two equal objects (as determined by =) have the same hash.

From: L. Peter Deutsch, Mark S. Miller
To: Smalltalk implementors group
Subject: Proposal for 1987 Smalltalk Standard meta-level interfaces to object state

This is a working document proposing a complete definition of the standard protocols for meta-level access to object state and to executable code for the 1987 version of the Smalltalk-80 Virtual Machine. The proposal includes a clean separation of object-level and meta-level aspects of Smalltalk consistent with both security and meta-interpretive definition, and without a significant change in the style of the language or environment. In order to do so, we include a coherent design for transparent forwarding and non-methodical objects as well (note: the implementability of this proposal doesn't depend on actually implementing non-methodical objects, they are introduced for explanatory clarity). Comments are solicited. For changes in successive versions, consult the version history.

This document is the first of a pair specifying meta-level interfaces for the 1987 Smalltalk standard. It is meant to be read first. The second document proposes a similar definition for meta-level access to control contexts and to processes.

Acknowledgements:

We wish to credit several important sources for the ideas presented here concerning meta-level facilities:

Ehud Shapiro & FCP for demonstrating & providing a model of how meta-interpretation provides a semantics yielding both debuggability & security.

Patty Maes for an encapsulated model of reflection.

Danny Bobrow for the generalization of MemoryAreas into nestable SecurityAreas.

Henry Lieberman for the insight that inheritance can be done through message passing.

And finally, Carl Hewitt, for the conviction that message passing among encapsulated actors is sufficient for everything.

Version history:

[3] ** in progress **: first version (derived from exec87.text.2): major contributions from Mark Miller, providing a first attempt at consistency with a secure meta-interpretive story, also a first spec for transparent forwarding and non-methodical objects.

The execution mechanism of the Smalltalk-80 Virtual Machine is defined by a virtual instruction set and a standard set of objects for representing execution state, and an interpreter that operates on this instruction set and these objects. In place of this, we propose to define only the externally visible behavior of the execution mechanism, not a particular algorithm that executes a particular representation of code and uses a particular representation of dynamic execution state. Indeed, our approach even allows a single system to represent code and state in several different coexisting ways. In particular, we propose that the only entities in the system that know the representation of executable code (other than the compiler) be the classes of the code objects, and that the message protocol of these objects be the basis of portability for the rest of the system such as the debugger. We propose a similar arrangement for the representation of dynamic execution state (Context objects). In other words, we propose that each running Smalltalk system include:

- A VM foundation, not written in Smalltalk, whose detailed interfaces are not specified (except generically, in that it must eventually support executing programs written in the Smalltalk language, including all the primitive methods that form part of the language definition).
- A language implementation kernel, written in Smalltalk, consisting of some classes whose interfaces are partially specified, but whose implementations are not specified.
- A set of programming tools (e.g. Browser, Debugger, Explainer), written in Smalltalk, whose functionality is not part of the VM definition.

[3] We strongly encourage vendors to implement their tools using only the standard specified interfaces of the language implementation kernel.

Meta-level separation (new for [3])

Recent research in the Lisp and Logic Programming worlds has established the benefit of implementing debugging and monitoring functions in terms of a "meta-interpreter". The idea is that even though all the execution structures of the language can be represented within the language itself, there is an explicit distinction between USING these structures implicitly and operating ON the structures explicitly. For example, Brian Smith's Reflective Lisp systems provide a linguistic mechanism for shifting between levels at any time; Udi Shapiro's work on Concurrent Prolog establishes a fixed boundary between levels, which appears preferable for protection purposes.

In this document we propose a coherent separation between meta-functionality and the normal world of Smalltalk execution, using a style closer to that of Concurrent Prolog. We introduce two interacting types of meta-level access: access to structure, and access to process. Meta-level access to structure means access to the internal state of an object, as if from the point of view of an interpreter executing a method that can access the object's state directly. This allows a clean semantics for inspectors: ordinarily, access to an object can occur only through its external protocol, but an inspector must access the object's state in the same way as an interpreter. We reify this machinery by means of a MetaObject, whose external protocol gives access to the internal state of the object that the MetaObject holds.

In both Concurrent Logic Programming and Actor systems, the notion of object and process are unified, so this kind of meta-level access is sufficient for purposes of debugging. However, in Smalltalk, an object does not have its own thread of control. A Smalltalk process cannot be seen as simply a meta-interpreter which is implementing method execution for the objects it is interacting with, because multiple processes may need to view the same object from a meta-level. So just as we create MetaObjects to give controlled access to the state of an object, we view ExecutionContexts (a redesign of the current Smalltalk-80 Contexts) as the meta-level objects through which we can view process.

Note that we separate these two types of meta-level access deliberately: access to process implies access to structure, but access to structure can be given while denying access to process.

In order to talk about execution in a way which is cleanly related to meta-interpretation, we need to do clean meta-interpretation. In order to support the Smalltalk style of environment, objects being executed at one meta-level need to be able to talk to objects being executed at another. In order to accomplish this, we need to introduce some new language constructs to talk about the interpretation process. While these are motivated by explanatory purposes, the functionality they provide must exist at some level of any actual Smalltalk system that has the semantics we propose, and we therefore propose that they be included in the language.

Scope of definition

The classes whose protocol we are concerned with defining, and the general nature of the protocol which we will specify for each, are:

Structure:

- Access to state:
  - MetaObject
    - meta-level access to original object
  - SecurityArea
    - find metaObject for a given object
- Access to description:
  - ClassDescription
    - save and retrieve source code
    - compile from source to executable code
    - queries of various kinds
  - ExecutableMethod
    - locate the source
    - queries
- Message handling:
  - NonMethodicalObject
    - syntax & informal semantics
    - relation to blocks
  - ApparentForwarder
    - interaction with remote object
    - relation to metaObjects

Process:

- ExecutionContext
  - step, proceed, restart, return
  - get the method
  - follow the enclosure (home) links
  - examine and change the named variables
  - correlate execution point with source code
- Process
  - create, suspend, continue, kill
  - examine and change priority

In addition, we will be somewhat concerned with classes that interact with these, such as BlockClosure and Semaphore.

In the descriptions below, messages with no other specified return value return their receiver.

Object, MetaObject, MemoryArea, and SecurityArea (rewritten for [3])
==========

As noted above, the direct instance variable access messages instVarAt: and instVarAt:put: are not compatible with a robustly protected system. Instead, we take the view that we do not expect an object to provide a de-encapsulated view of itself; rather, we introduce the notion of a meta-object that provides a view of an object as if from the privileged point of view of an interpreter executing methods that operate on the object's state directly. We must also restrict the creation of meta-objects in a way that allows for protection. We choose to do this by reifying the notion of a "security area", which represents permission to deal with a particular set of objects from a meta-level view. It is only this security area which can convert a reference to an object into a reference to its meta-object. Given just a reference to an object, but without reference to its security area, we intend that there be no way to gain access to the meta-object.

<MetaObject | nil> := <SecurityArea> metaObjectFor: <Object>

A return value of nil means that this SecurityArea doesn't represent meta-level permission for this object.

The Smalltalk kernel only provides a flat space of SecurityAreas -- one for each physical object memory system. (An ordinary single-user Smalltalk system will probably have only one object memory system, or perhaps a second one for handling out-of-memory emergencies.) Note that the user may easily define new classes of SecurityAreas which are subsets or unions of those provided by the kernel.

<Object> := <MetaObject> subject

This message returns the original object that this meta-object is a meta-object for.

<Behavior> := <MetaObject> classOfSubject

Returns the actual class of <MetaObject subject>. Note that this is not necessarily the same as <subject class>, as <subject class> returns only what the object externally claims is its class. In particular, if we ask a transparent forwarder what its class is, we find the class of the object it is forwarding to. If we ask its meta-object what its classOfSubject is, we get NonMethodicalObject (to be introduced later).

<Dictionary> := <MetaObject> instVarDictionary

This message returns an object that appears to be a Dictionary, in which the keys are the original receiver object's instance variable names (if the object has named instance variables) or an appropriate range of integers (if the object has indexed instance variables). This dictionary responds to the full range of Dictionary protocol: thus, for example, it provides for finding the value of any instance variable (using at:), setting the value of any instance variable (using at:put:), enumerating the instance variables (using do: or keysDo: or associationsDo:), etc. Note that the dictionary continues to present the current state of the object, not a snapshot. Needless to say, this "dictionary" is not an instance of the standard Dictionary class, it merely implements Dictionary protocol.

We assume that an object's SecurityArea has unconstrained access to the object memory system that stores the state of the object. Therefore, an object's meta-object and instVarDictionary access (and change) this state by sending messages to the object's SecurityArea. We do not attempt, in this document, to standardize the protocol with which this interaction takes place.

One would like the Smalltalk "dependents" mechanism to be usable with meta-objects and instance variable access dictionaries, so that an object could be notified whenever the state of another object changed at the instance variable level. (The dependency mechanism is not part of the proposed standard, but is implemented in many existing Smalltalk systems.) While there is no logical obstacle to this, we recognize that implementing this may be difficult or impossible in some implementations of the Smalltalk-80 virtual machine. Therefore, we encourage the implementors of those systems that do provide a dependency mechanism to either implement it for meta-objects and instance variable access dictionaries, or to reject attempts to add dependents to such objects.

ClassDescription / Behavior

Unlike the present system, our proposal here does not make manifest any Compiler or Parser class. Instead, each class is responsible for compiling methods defined within it. This allows different classes to use different source languages.

Compiling

<ExecutableMethod> := <Behavior> compile: <String> in: <NameScope> ifError: <BlockClosure>
Compile a piece of source text and return an ExecutableMethod. If an error occurs, apply the BlockClosure to an "error object" argument. The error object must respond to at least the following messages:

string - return an error message string;
position - return a source position for indicating where the error occurred.
Implementations may choose to include further information in the error object.

Properties of ExecutableMethods are described in more detail below.

Accessing

The following messages from chapter 16 of the Blue Book are hereby standardized:
<Behavior> := <Behavior> superclass
<Behavior> superclass: <Behavior>
We would like to provide the complementary message
<(Collection of: Behavior)> := <Behavior> subclasses
Unfortunately, this is conceptually incompatible with distributed systems, systems in which only part of the code is loaded into the machine's address space (or fast retrieval space), etc. Another conceptual objection is that specializing a class should not result in a change in behavior of the class being specialized. The only solution we presently see for this problem is to provide, at the user-level (as opposed to the kernel, and therefore not part of this document), subclass registry dictionaries local to e.g. a specific user. When that user makes a subclass, he registers this with his local subclass registry. When he wants to find out what subclasses of a given class HE KNOWS ABOUT, he ask his local registry. We have not experimented with this idea, and it is clearly less convenient than the present arrangement.

The following messages are added or changed:
<(Collection of: Symbol)> := <Behavior> instVarNames
Return a Collection of the receiver's instance variable names. If the receiver has indexed instance variables, the result is a collection with one of the special symbols 'Object*' asSymbol or 'Byte*' asSymbol, compatible with the syntax for creating classes.
<Dictionary> := <Behavior>classVarDictionary
Returns a Dictionary of the receiver's class variables.
<Dictionary> := <Behavior> methodDictionary
Return a Dictionary-like object representing the methods known to this class. This object is probably not exactly a MethodDictionary object as in the current Smalltalk-80 system: for example, it may need to remember the class it came from, for selective cache invalidation when a new method is added. Again, the philosophy is to present a Dictionary-like interface, with the implementation left up to the system implementor. This interface replaces specialized accessing messages such as selectors (= methodDictionary keys), compiledMethodAt: (= methodDictionary at:), etc.

Note that all Behaviors, not just Classes, are specified as having instance and class variables.

We specifically do not propose to standardize any Behavior protocol regarding non-linguistic information such as organization categories, comments, subclass registries, etc.

Querying

Querying messages fall into two groups: those that return a set of methods with some property, and those that return a set of values obtained from all the methods in the class. The querying messages referring to variables only refer to those visible at the class level, i.e. the instance and class variables of the receiver class and its superclasses, plus the (read-only) pseudo-variables 'self' and 'super'.

<(Collection of: Symbol)> := <Behavior> whichMethodsRead: <Symbol>
<(Collection of: Symbol)> := <Behavior> whichMethodsWrite: <Symbol>
<(Collection of: Symbol)> := <Behavior> whichMethodsReadOrWrite: <Symbol>
Return the selectors of the methods which respectively read, write, or either read or write the

variable with the specified name. If there is no such variable visible from this class, return an empty collection.

<(Collection of: Symbol)> := <Behavior> whichMethodsSend: <Symbol>
Return the selectors of the methods which send the specified message. We explicitly require that this message, and its companion messagesSent, give accurate answers with reference to the source code: in particular, the answer must include messages sent at the source level that may not be explicitly present in the object code (such as ifTrue:), and must not include messages present in the object code that implement language features (such as blockCopy: in the present Smalltalk-80 system). Such exceptions might be handled with a small bit table in each ExecutableMethod, or with extra entries in a literal table, or in other ways. For performance reasons, we strongly discourage implementors from implementing this message (or any other query message) by reanalyzing the source code.

<(Collection of: Symbol)> := <Behavior> whichMethodsUseLiteral: <Object>
Return the selectors of the methods which refer to the specified object as a literal object. Symbols that are only used as selectors are NOT included in this query.

<(Collection of: Symbol)> := <Behavior> variablesRead
<(Collection of: Symbol)> := <Behavior> variablesWritten
<(Collection of: Symbol)> := <Behavior> variablesReadOrWritten
Return the names of the variables read, written, or either read or written by any method within the class.

<(Collection of: Symbol)> := <Behavior> messagesSent
Return the selectors of all messages sent by any method in this class.

<(Collection of: Object)> := <Behavior> literalsUsed
Return all the literals used by any method in this class.

ExecutableMethod

We explicitly take the position that every ExecutableMethod be able to access (or regenerate) its source code, and to report the class in which it was compiled and the selector it is associated with. (The latter follows from the ability to retrieve the source code.) The former capability is present in the current Smalltalk-80 system; the latter is not. We expect that most systems will store both pieces of information directly in the ExecutableMethod in some form. Note that we are not requiring that ExecutableMethods be the "inner" representation of compiled code in the Smalltalk system, only that they appear in the places specified by this document.

Accessing

<Behavior> := <ExecutableMethod> definingClass
Return the class where the method is defined.

<SmallInteger> := <ExecutableMethod> numArgs
Return the number of arguments that the method expects.

<Symbol | nil> := <ExecutableMethod> selector
Return the selector this method corresponds to, unless the method is the method of a non-methodical object (see below), in which case return nil.

<Object> := <ExecutableMethod> source
Return an object representing the source code for the method. The nature of the object is undefined: it need only be acceptable as the source argument in the compile:... method for the defining class. We do specify that in every system that includes a compiler for the standard source language, the compiler must accept Strings as input (i.e. objects that recognize String protocol and respond to the message asString by returning an appropriate String, such as String, Text, and Paragraph in the current Smalltalk-80 system.)

Querying

All the information-retrieval querying messages applicable to classes (but not the method-retrieval queries) are applicable to individual ExecutableMethods, namely:

<(Collection of: Symbol)> := <ExecutableMethod> variablesRead
<(Collection of: Symbol)> := <ExecutableMethod> variablesWritten
<(Collection of: Symbol)> := <ExecutableMethod> variablesReadOrWritten
<(Collection of: Symbol)> := <ExecutableMethod> messagesSent
<(Collection of: Object)> := <ExecutableMethod> literalsUsed

These queries only return the information obtained from the individual method. The visible variables are precisely those visible from within the method, i.e. those visible at the class level, plus the argument and temporary variables of the method (but not of any blocks), 'self', and the pseudo-variable 'super'.

We encourage implementors to implement these messages by returning a special kind of collection that extracts requested information from the ExecutableMethod on demand, rather than (for example) copying the entire response into an Array at the time the query message is received.

NonMethodicalObject

A non-methodical object is an object that can process with one method any message sent to it (as opposed to doing method lookup). It does exactly what objects that respond only to "doesNotUnderstand:" approximate. We propose the following Block-like syntax for creating non-methodical objects:

[::message | rest-of-a-block-after-the-parameters]

or, in the meta-syntax of the new syntax document:

non-methodical-object = '[' '::' declared-variable-name '|' [temporaries] statements ']'
primary = ... | non-methodical-object

Non-methodical objects are block-like closures, with all the same scoping rules as for blocks. They are also created in an identical fashion: by evaluating a non-methodical-object expression. The difference is in their response to messages. A non-methodical object responds to any message by binding the message itself to its one parameter variable and invoking its body.

The reason for introducing a syntax for creating non-methodical objects, rather than simply using the normal methods for creating an instance of class NonMethodicalObject, is that, like blocks, and unlike ordinary objects, they must be able to directly share variables already existing at the time the non-methodical object is created. (We hope that some future Smalltalk language will provide this ability for objects in general, but we don't have any proposal to make in this area at the moment.) Also, it must be possible to create and initialize a non-methodical object in a single atomic action, since one cannot send it initialization messages.

We also introduce the following syntax for sending computed messages:

<Object> :: <Object>

or, in the meta-syntax of the new syntax document:

keyword-message = ... | ('::' primary binary-message*)+

Note that '::' has the precedence of a keyword selector. The above expression does what "<Object> performMessage: <Message>" approximates: it sends the object on the right as a message to the object on the left. In any context and for any expr and body "[::x | body] :: expr" is equivalent to "[:x | body] value: expr". It is of course an error to send anything other than a <Message> to a normal, methodical

object. NonMethodicalObject and BlockClosure are both subclasses of Closure.

Again, since a non-methodical object must be able to handle EVERY message identically, we need a syntactic construct to express sending a message directly to an object: we cannot use a message for this purpose.

As NonMethodicalObjects do not perform method lookup in response to receiving a message, the class "NonMethodicalObject" should define no methods for its instances, and indeed must be distinguished syntactically in some way. A plausible implementation of NonMethodicalObjects as an incremental change to many systems is for the methodDictionary instance variable of the class NonMethodicalObject to contain something which is manifestly not an instance of MethodDictionary. Checking for this, which is only necessary when the various lookup caches fail and a full lookup is required, should not have significant performance impact.

ApparentForwarder
==========

An ApparentForwarder is typically the intermediary between a transparent forwarder and the object it is forwarding to. It responds to the following protocol:

```
<ApparentForwarder> := <ApparentForwarder class> subject: <Object>
<Object> := <ApparentForwarder> subject
<Object> := <ApparentForwarder> valueForMessage: <Object>
```

The third expression should be equivalent to:

```
<Object> := <ApparentForwarder> subject :: <Object>
```

A transparent forwarder can easily be set up to "front-end" for an ApparentForwarder as follows:

```
<ApparentForwarder> transparentForwarder
    ^ [ ::msg | self valueForMessage: msg ].
```

An apparent forwarder for an object which is actually on a remote machine cannot locally respond with the true subject, but it can respond to the "subject" message with a transparent forwarder which is front-ending for itself. This is why the above code uses "valueForMessage:" instead of "... subject :: ... " despite our stated behavioral equivalence. This equivalence only applies "on the outside". We apply this technique below:

MetaObject is a subclass of ApparentForwarder, with the subject being the object that the metaObject represents. It is consistent to view any object as just a transparent forwarder for its own metaObject, with all the work for responding to a message being contained the metaObject's valueForMessage: method. In some sense, the metaObject is forwarding the message to the idea of the object specified by its implementation, by bringing about the behavior specified by this implementation. In particular, the valueForMessage: method for a metaObject of a methodical object might be as follows:

```
<MetaMethodicalObject> methodForMessage: aMessage ifFail: failureBlock
    | selector class method |
    selector _ aMessage selector.
    class _ self classOfSubject.
    ^ class methodDictionary at: selector ifFail: failureBlock.

<MetaMethodicalObject> argsForMessage aMessage
    ^ aMessage args

<MetaObject> valueForMessage: aMessage
    | method args |
```

```
        method _ self methodForMessage: aMessage ifFail: [...].
        args _ self argsForMessage aMessage.
        ^ method valueForMetaReceiver: self arguments: args
```

An attempt to actually implement everything this way would cause one to infinitely regress up the infinite tower, but the system must act in a way consistent with this story. (It is allowed for the system to deviate from the story to the extent of executing within finite memory and time)

It is assumed above that the dictionary returned by "class methodDictionary" will search up the superclass chain in looking up a method. Note that since an ExecutableMethod knows its definingClass, it can process sends to 'super' without being told what class it was found in, e.g.:

```
    <ExecutableMethod> valueForMetaReceiver: aMetaObject sendSuper: aMessage
        | selector args class method |
        selector _ aMessage selector.
        args _ aMessage args.
        class _ self definingClass superclass.
        method _ class methodDictionary at: selector ifFail: [...].
        ^ method valueForMetaReceiver: self arguments: args
```

In order to implement the current Smalltalk semantics, the ifFail: blocks above should send doesNotUnderstand: messages to the original receiver. In order to address the "asArkObject" problem, we suggest that instead a "receiver: <original receiver> doesNotUnderstand: <message>" be sent to the selector. The method for this in class Symbol can in turn implement the current functionality.
}

Execution

```
[M-]{
        <Object> := <ExecutableMethod> valueForReceiver: <Object> arguments: <Array>
}
[M+]{
        <Object> := <ExecutableMethod> valueForMetaReceiver: <MetaObject> arguments: <Array>
}
```

Run the method with the given receiver and arguments, and return the resulting value. If [M-]{the object} [M+]{"<MetaObject> classOfSubject"} is not of a compatible class (i.e. the method's defining class or a sub...subclass), or the number of arguments is wrong, an error occurs. This message is provided mostly for 'doIt' in the interactive interface: BlockClosures are intended to be the objects normally used to represent executable procedures. Note that just as for any other send, there is no "unwind protection" in case a block does an ^ return. [M?]{I don't understand the comment about "unwind protection".}

[M+]{
Note that "arguments: <Array>" above is an <Array> of regular objects, not metaObjects. The <ExecutableMethod> needs priviledged access to the receiver, typically in order to "at:" and "at:put:" its instVarDictionary in the meta-interpretive story. However, it only needs to send messages to the other arguments. This is consistent with the story that the other objects are also being meta-interpreted in the same way: in this story these messages are just getting sent to transparent forwarders which are sending "valueForMessage:" messages to corresponding metaObjects which look up methods ...

By <MetaObject> and <ExecutableMethod>, we of course mean abstract data types corresponding to the protocols specified in this document. If both the <ExecutableMethod> and <MetaObject> above are implemented specially by the kernel, then this can execute without further messages being exchanged between them. (Indeed, this must be so to avoid infinite regress) However, if either is a non-special object which satisfies its abstract protocol, the other must interact with it properly according to this

protocol. This may be hard for primitive methods.

From: L. Peter Deutsch, Mark S. Miller
To: Smalltalk implementors group
Subject: Proposal for 1987 Smalltalk Standard meta-level interfaces to contexts and processes

This is a working document proposing a complete definition of the standard protocols for meta-level access to execution contexts and processes for the 1987 version of the Smalltalk-80 Virtual Machine. The proposal includes a clean separation of object-level and meta-level aspects of Smalltalk consistent with both security and meta-interpretive definition, and without a significant change in the style of the language or environment. Comments are solicited. For changes in successive versions, consult the version history.

This document is the second of a pair specifying meta-level interfaces for the 1987 Smalltalk standard. The first document proposes a similar definition for meta-level access to object state and to executable code. You should read the other document first: you are likely to find the concepts in this one unfamiliar and confusing if you don't.

Acknowledgements: see the first document.

Version history:

[3] ** in progress **: first version (derived from exec87.text.2): major contributions from Mark Miller, providing a first attempt at consistency with a secure meta-interpretive story, also a first spec for transparent forwarding and non-methodical objects.

[M-]{
BlockClosure
==========

A BlockClosure conceptually contains just two pieces of information: a home context, used for accessing outer-scope variables and for non-local returns, and a method to execute when the closure is applied to arguments. An implementation may choose to represent this information in some other form, e.g. it may normally represent the method using the Smalltalk-80 device of embedding the block code inside its home method, but the logical view must be as described. In particular, the block must appear to have its own ExecutableMethod, conceptually distinct from the ExecutableMethod for the enclosing method. The ExecutableMethod for a block responds to the messages described above in the following way:

- The source code is the source code for just the interior of the block, not the entire enclosing method.
- The query messages refer only to the code within the block. The visible variables are precisely those visible from within the block, i.e. all those visible from the method, plus the argument and temporary variables of the block and any enclosing blocks.
- The message valueForReceiver:arguments: invokes the block. The receiver (first) argument must be an appropriate BlockClosure.

When we say the block must "appear to have" its own ExecutableMethod, we do not rule out (indeed, we encourage) implementations in which the block's "ExecutableMethod" is just a mediator object that implements ExecutableMethod protocol by dynamically extracting information from the enclosing method's ExecutableMethod. This option illustrates the flexibility of Smalltalk's totally object-oriented approach to system design.

Accessing

<ExecutableMethod> := <BlockClosure> method

Return the ExecutableMethod for the block. As explained above, this must be a full-fledged method in terms of its external protocol, conceptually distinct from the ExecutableMethod for the enclosing method.

<ExecutionContext> := <BlockClosure> outerScope

Return the next outer scope of the block.

<ExecutionContext> := <BlockClosure> remoteReturnContext

Return the context to which control would return if the block code did an ^ return. In current Smalltalk-80 terminology, this would be the sender context of the block's home.

If we decide to adopt a continuation model for control, a BlockClosure actually has two different 'homes': one that holds the outer-scope variables, and one that represents the continuation for ^ returns. For this reason, as well as the conceptual distinction, we have treated these as though they were different objects. [M?]{ Even if we don't adopt a continuation model, these two are different. The outerScope could be the containing BlockExecutionContext, whereas the remoteReturnContext would still be a MethodExecutionContext.}
}

[M+]{
MetaClosure
==========

A MetaClosure is a metaObject for either type of closure: BlockClosures or NonMethodicalObjects. Note that closures themselves are not considered meta-level objects.

A Closure conceptually contains just two pieces of information: a home context, used for accessing outer-scope variables and for non-local returns, and a method to execute when the closure is applied to arguments. An implementation may choose to represent this information in some other form, e.g. it may normally represent the method using the Smalltalk-80 device of embedding the closure code inside its home method, but the logical view must be as described. In particular, the closure must appear to have its own ExecutableMethod, conceptually distinct from the ExecutableMethod for the enclosing method. The ExecutableMethod for a closure responds to the messages described above in the following way:

- The source code is the source code for just the interior of the closure, not the entire enclosing method.
- The query messages refer only to the code within the closure. The visible variables are precisely those visible from within the closure, i.e. all those visible from the method, plus the argument and temporary variables of the closure and any enclosing Contexts.

- The message valueForMetaReceiver:arguments: invokes the closure. The receiver (first) argument must be an appropriate MetaClosure.

When we say the closure must "appear to have" its own ExecutableMethod, we do not rule out (indeed, we encourage) implementations in which the closure's "ExecutableMethod" is just a mediator object that implements ExecutableMethod protocol by dynamically extracting information from the enclosing method's ExecutableMethod. This option illustrates the flexibility of Smalltalk's totally object-oriented approach to system design.

Accessing

<ExecutableMethod> := <MetaClosure> method

Return the ExecutableMethod for the closure. As explained above, this must be a full-fledged method in terms of its external protocol, conceptually distinct from the ExecutableMethod for the enclosing method.

<Dictionary> := <MetaClosure> outerScope

A closure may refer freely to variables defined in its lexically enclosing defining context. The closure must retain, as part of its state, the bindings of these variables created by the activations that resulted in the creation of this closure. The dictionary refered to above represents these bindings, through it the values of any of these variables may be retrieved or changed. Note that this dictionary is only obligated to represent binding of variables which are actually refered to freely in the method of the closure (although it may represent all lexically apparent binding if it wishes). We allow it this freedom in order to not interfere with some important optimizations associated with closure in the Scheme world. (It may even be necessary to only guarantee access if the original closure method uses the variable as an rValue (evaluated for its value), and only guarantee the ability to change the variable if the original method uses it as an lValue (target of assignment))

Note that the dictionary remembers the "bindings" of the variables, not simply their values. This means that if two closures are created by the same activation and refer to the variable, that changes to its value made by one are visible to the other.

[M?]{
I don't know why we need access to a remoteReturnContext, but if we do, we should return a continuation.

<Continuation> := <MetaClosure> remoteReturnContext

A continuation is not a meta-level construct, and yields no meta-level view. This makes continuations very different from executionContexts. Until and unless we define the mechanics of continuations, I suggest we completely leave this message out.
}

MetaNonMethodicalObject is a subclass of MetaClosure, and is the class for all metaObjects of nonMethodicalObjects. There is no new protocol defined by this class, it is introduced so it can override the "methodForMessage:" and "argsForMessage:" methods as follows:

```
<MetaNonMethodicalObject> methodForMessage: aMessage
    ^ self method

<MetaNonMethodicalObject> argsForMessage: aMessage
    ^ Array with: aMessage
}
```

[M+]{
BlockClosure
==========
}

Invocation

<Object> := <BlockClosure> value
<Object> := <BlockClosure> value: <Object>
<Object> := <BlockClosure> value: <Object> value: <Object>
<Object> := <BlockClosure> value: <Object> value: <Object> value: <Object>
<Object> := <BlockClosure> value: <Object> value: <Object> value: <Object> value: <Object>
(etc.)
<Object> := <BlockClosure> valueWithArguments: <Array>
Invoke the closure with the indicated arguments. Any given implementation may choose to

provide the value:...value: form for more than 4 arguments ad lib. If the wrong number of arguments is supplied, an error occurs.

Control structures

The looping constructs of the language are implemented as messages to BlockClosure; they may also be treated specially by the compiler.

<BlockClosure> whileTrue: <BlockClosure>
Invoke the receiver; as long as the result is true, apply the argument, then apply the receiver again. If the result of invoking the receiver is neither true nor false, an error occurs.

<BlockClosure> whileFalse: <BlockClosure>
Invoke the receiver; as long as the result is false, apply the argument, then apply the receiver again. If the result of invoking the receiver is neither true nor false, an error occurs.

<BlockClosure> whileTrue
Equivalent to <BlockClosure> whileTrue: [].

<BlockClosure> whileFalse
Equivalent to <BlockClosure> whileFalse: [].

<BlockClosure> repeat
Equivalent to [true] whileTrue: <BlockClosure>.

ExecutionContext / MethodContext / BlockContext
==========

We reaffirm our commitment to the Smalltalk object-oriented model of execution by proposing not to compromise the unrestricted object-oriented model of accessing execution state. This does not rule out implementations where some more efficient implementation (such as linear stacks) is used most of the time: we only require that the implementation present the following object-oriented interface.

Accessing

<Dictionary> := <ExecutionContext> localVarDictionary
Return a Dictionary-like object for accessing the local (argument and temporary) variables defined in this context, similar to <Object> instVarDictionary. The dictionary covers only variables defined in the receiver context, not in any of its outer scopes: the caller must step through these explicitly using the outerScope message. The[M-]{ name 'thisContext' is recognized in every context; the} name 'self' is recognized only in MethodContexts, and is read-only (not valid for at:put:). 'super' is a language construct, not a variable name.

<ExecutionContext> := <ExecutionContext> sender
Return the context to which this context would return. For BlockContexts, this is the local (fall-off-the-end) return, not the ^ return.

<ExecutableMethod> := <ExecutionContext> method
Return the method which this context is executing.

[M-]{

<Object> := <ExecutionContext> receiver
Return the receiver of the message that created this context. For BlockContexts, this is the BlockClosure, since the BlockClosure was the receiver of the value or value:* message.

}

[M+]{

<MetaObject> := <ExecutionContext> metaReceiver
Return the metaObject of the receiver of the message that created this context. For BlockContexts, this is the MetaBlockClosure, since the BlockClosure was the receiver of the value or

value:* message.
]

<Integer> := <ExecutionContext> sourcePosition
Return the character position in the source code which corresponds to the current execution point in the context. If the context is about to send a message, the position will indicate the first character of the selector; if the context is about to return, the position will indicate the closing] of a block, or just past the end of a ^ statement, or just past the last character of the method; otherwise, the position indicates the approximate locus of control (for example, just after a .). [M?]{This is specific to the standard source language. This should be stated, as well as what requirements must be met independent of source language.}

Execution

All the messages having to do with execution are messages to the process, not to the context. We propose this because there may be implementation-dependent state (such as saved registers) that individual contexts may not be able to locate or update in general. [M?]{ I don't understand this. Isn't a context specific to a given process? Doesn't it know what process it's specific to? If it does, then it can access this state from its process. Therefore, the decision as to whether to locate some specific functionality with the process or the context should be based on expressiveness.}

[M+]{

Process level meta-interpreter

Processes are considered meta-level objects, not because of concurrency, but simply because they are a means of controlling execution. We would like to see these two issues better separated, but have not yet done so ourselves. (A place to look is the Logix operating system: "computations" are the means for meta-interpretively controlling execution, but a single "computation" may contain zillions of processes.)

A process needs to control execution at the granularity of individual message sends (but not variable accesses). We need to enhance our meta-interpretive story so that the Process object is responsible for carrying out each send operation of the execution it is animating. In order to understand this issue, we present an explanatory implementation for "<Process> step". A running process is one which is sending "step" messages to itself in a loop.

```
<Process> step
    "Note: this code doesn't deal with Semaphores or Waiting"
    | nextStepAction currentContext nextContext |
    nextStepAction _ self nextStepAction.
    currentContext _ self currentContext.
    nextStepAction = #return
            ifTrue: [nextContext _ currentContext sender]
        ifFalse: [nextStepAction = #advance
            ifTrue: [nextContext _ currentContext advance]
        ifFalse: [nextStepAction = #send
            ifTrue: [nextContext _ self stepSend: currentContext]]]
    nextContext isNil
        ifTrue: [state _ #Dead]
        ifFalse: [self currentContext: nextContext]

<Process> stepSend: currentContext
    | reciever metaReciever message method args |
    reciever _ currentContext recieverOfSend.
```

```
        message _ currentContext messageToBeSent.
        currentContext advance.
        metaReciever _ securityArea metaObjectFor: reciever.
            "Note: this implies that a securityArea is
                part of the state of a process"

        metaReciever isNil
            ifTrue: [ "This process doesn't have meta-level
                        permission for the target object, so
                        just send the message normally"
                    ^ reciever :: message]
        method _ metaReciever methodForMessage: message.
        args _ metaReciever argsForMessage: message.
        ^ Context
            sender: currentContext
            metaReceiver: metaReciever
            method: method
            args: args.

    <Context class> sender: sender metaReceiver: metaReceiver method: method args: args
        | params temps localVarDictionary |
        params _ method params.
        temps _ method temps.
        localVarDictionary _ Dictionary new.
        args size = params size ifFalse: ["indicate error" ...]
        1 to: args size do: [ :i | localVarDictionary at: (params at: i) put: (args at: i)].
            "is there any better way to map down two lists in parallel?"
        temps do: [ :temp | localVarDictionary at: temp put: nil ].
        ^ super new
            sender: sender
            metaReceiver: metaReceiver
            method: method
            localVarDictionary: localVarDictionary
            pc: 0

    <Context> advance
        pc _ method pcAfter: pc
            withMetaReceiver: self metaReceiver
            withBindings: self localVarDictionary
        ^ self

    <Context> recieverOfSend
        ^ method evalReceiverOfSendAt: pc
            withMetaReceiver: self metaReceiver
            withBindings: self localVarDictionary

    <Context> messageToBeSent
        ^ method evalMessageAt: pc
            withMetaReceiver: self metaReceiver
            withBindings: self localVarDictionary

    <MetaObject> argsForMessage: message
        ^ message args

    <MetaNonMethodicalObject> argsForMessage: message
        ^ Array with: message
```

```
<Process> nextStepAction
    ^ self currentContext nextStepAction

<Context> nextStepAction
    ^ method stepActionAt: pc
```

This demands that <ExecutableMethod>s respond to the following protocol:

```
<Symbol> := <ExecutableMethod> stepActionAt: <pc>
```

For a given pc, if the above returns #advance or #send, then:

```
<pc> := <ExecutableMethod> pcAfter: <pc>
```

returns the pc of the next block of code after the given pc.

If the above returns #send, then:

```
<Object> evalReceiverOfSendAt: <pc> withMetaReceiver: <MetaObject> withBindings: <Dictionary>
<Object> evalMessageAt: <pc> withMetaReceiver: <MetaObject> withBindings: <Dictionary>
```

This meta-interpreter doesn't deal with sends to 'super'. In order to do so cleanly, we probably need a #sendSuper in addition to the above types of stepAction. It also doesn't deal with the nested evaluation of arguments. Peter informs me that an attempt to do so for Interlisp proved difficult. This should probably wait for a continuation passing story.

This meta-interpreter is not complete, and is probably not consistent. It would be interesting to actually get such a meta-interpreter working. We will then probably understand the semantics of what we are doing much better.

}

Proposal for 1987 Smalltalk Standard: Processes

prepared by:
L. Peter Deutsch, Xerox, ParcPlace Systems

other contributors (alphabetical order, partial listing):
Mark Miller, Xerox PARC/ISL
Allen Wirfs-Brock, Tektronix

This is a working document specifying the standard behavior of Processes and Semaphores for the 1987 version of the Smalltalk-80 language. Comments are solicited.

Version history:

[1] 25 February 1987: first version, split off from exec87.text. This is the version distributed at Feb. 26-27 implementors workshop.

Process

Processes are lightweight threads of control, as in Smalltalk-80 and many Lisp systems: they are not protection domains or separate address spaces, as in Unix, or separate naming environments or explicit linear stack areas, as in ZetaLisp. (This is not meant to imply that such facilities are not useful, only that the term "process" as used in this document, and the Smalltalk standard class Process, do not cover them.)

From the client's point of view, a process has just a few pieces of visible state:

- A scheduling state: Dead, Suspended, Waiting <Semaphore>, WaitingSuspended <Semaphore>, or Active.
- A numeric priority, a positive integer with an implementation-dependent maximum value. Processes with higher priority run in preference to those with lower priority: see below for more details. We require that implementations provide at least 8 priority levels.
- An execution state, consisting of a Context and possibly some implementation-dependent information such as saved registers. Each of these state components is the subject of sections below.

In the current Smalltalk-80 system, there is a ProcessorScheduler that holds the priority lists of runnable Processes, and also handles the fine-grained timer. Since Smalltalk appears to be a promising language for multiprocessors, we wish to specify its process facilities in a way that does not limit it to execution on a uniprocessor. For this reason, we deliberately choose not to define a Processor or ProcessorScheduler object; instead, we reassign the functions of the current

ProcessorScheduler to class Process, except for the timer function which we recommend be given to a new Timer object.

Scheduling states

From the client's point of view, a process is in one of four scheduling states:
- Dead: it cannot be run. The only time a process is dead is after it has terminated.
- Suspended: it is potentially runnable, but it must be explicitly resumed.
- Waiting: it can be run as soon as a particular Semaphore is signalled.
- WaitingSuspended: it can be run as soon as a particular Semaphore is signalled and it is explicitly resumed.
- Active: it can run at any time, according to the priority decisions made by the scheduler. This includes any processes that are running now (in a multi-processor system, there may be more than one.)

The transitions between these scheduling states occur as follows:
Active -> Waiting: the process sends a wait message to a Semaphore with no signals.
Active -> Suspended: the process receives a suspend message.
Waiting -> WaitingSuspended: the process receives a suspend message.
Active -> Dead: the process returns from its root context.
(Active, Waiting, Suspended, WaitingSuspended) -> Dead: the process receives a kill message.
Waiting -> Active: the process is waiting on a Semaphore, and the Semaphore receives a signal message.
Suspended -> Active: the process receives a resume message.
WaitingSuspended -> Waiting: the process receives a resume message.
WaitingSuspended -> Suspended: the process is waiting on a Semaphore, and the Semaphore receives a signal message.
Dead -> Suspended: the process' execution state is reset.

We have deliberately omitted any distinction between processes that CAN run and processes that ARE running (aside from the ability for a running process to find out its own identity), and any facilities for querying the scheduler as to the current set of Active processes. This implies, in particular, that the Smalltalk environment must explicitly keep track of which process will be interrupted if the user types a control-C. We consider this an improvement over the current haphazard selection of a running process.

Priorities

It has been observed that the current Smalltalk-80 notion of process priorities doesn't work very well for any of its intended uses: the desire to provide fast response to I/O events (including user interaction) would better be served by a deadline mechanism, and the desire to allocate CPU shares would better be served by a bidding mechanism or some other "soft" priority arrangement. While we agree

with these observations, limited time has prevented us from attempting to specify a new scheduling mechanism.

The current Smalltalk-80 scheduler makes some strong guarantees about process priorities and non-preemption, on which a good bit of current software unfortunately depends. These guarantees are not compatible with multiprocessors, or even with most operating system schedulers. Therefore, we propose only to guarantee a weaker set of properties:

- If all running processes are at priority M or lower, and a process at priority N > M becomes ready, the latter process will run, and one of the running processes may be preempted.
- If a process stops running, and the highest priority process still running is at priority M, and a process at priority N > M is ready, the latter process will run, and one of the running processes may be preempted. This is not implied by the first property, since the first property only deals with what happens when a new process becomes ready.

Note that we do NOT guarantee the following, which is commonly assumed by current Smalltalk systems:

- As long as a process at priority M is running, no (other) process at priority N <= M will run.

(By convention, if there is no process running, we pretend there is a process running at non-existent priority 0.) In a multi-processor implementation, we cannot guarantee that the highest-priority processes will always be the ones running: it may simply be too expensive to exchange enough priority information between processors to always maintain this property.

Process protocols

Creation

<Process> := <Process class> forBlock: <BlockClosure> priority: <SmallInteger>
Create a Suspended process that will start execution in an activation of the closure when Activated. The process is killed when it returns from the context created for the closure. (In Mark Miller's meta-objects proposal, the first argument should be a MetaClosure, not a BlockClosure.)

Scheduling

<Process class> current
Return the currently running Process, i.e. the one whose execution sent this message.

<Process> resume
Make a Suspended process Active, and a WaitingSuspended one Waiting.

<Process> suspend
Suspend a process in any state except Dead. A Waiting process becomes WaitingSuspended.

<Process> kill
Make a process in any scheduling state Dead. A Dead process cannot be put into any other scheduling state until it has had its execution state reset.

<Symbol> := <Process> schedulingState
Return the scheduling state of a process: #active, #waiting, #suspended, #waitingSuspended, or #dead.

<Semaphore> := <Process> currentSemaphore
If the receiver is Waiting or WaitingSuspended, return the Semaphore on which the receiver is Waiting. If the receiver is not Waiting, return nil. Note that the Semaphore may be signalled at any time -- there is no guarantee that the process will still be Waiting the next time it is queried.

Priority

<Integer> := <Process> priority
Return the priority of a process.
<Process> priority: <Integer>
Set the priority of a process. Note that even if a process has a priority higher than all other processes, there is no guarantee that it will be the only running process: setting the priority to the highest value is not an adequate means to guarantee mutual exclusion.
<Integer> := <Process class> highestPriority
Return the highest acceptable priority value.

Semaphores

The semantics of Semaphores similar to those the current Smalltalk-80 system, except that we do not guarantee that processes waiting on a Semaphore will be activated in FIFO order: instead, we only guarantee that if a process P is waiting on a Semaphore, only a finite number of other processes waiting on the same Semaphore will be activated before P is activated. (This is called a "non-starvation" condition.)

<Semaphore> wait
If the receiver is holding any signals, decrement its signal count. If the receiver is not holding any signals, make Waiting the (Running) process sending the wait message.
<Semaphore> signal
If any processes are Waiting on the receiver, make one of them Active. If no processes are Waiting, increment the

excess signal count of the receiver. If a process is WaitingSuspended, make it Suspended.

<Semaphore> waitingProcesses

Return a collection of the processes Waiting on the receiver, if any. The order of processes in the collection, or even whether the collection is ordered, is not guaranteed. If there are no processes Waiting on the receiver, return an empty collection.

It has been observed that Semaphores are a very low-level synchronization mechanism, and that their use for I/O devices in particular leads to awkward arrangements where the synchronizing signal and the data transfer must be handled separately. Two alternatives have been proposed: request/reply, as in the V Kernel (an experimental lightweight operating system kernel being developed at Stanford), or monitors or serialized objects. We have tentatively rejected request/reply because its semantics are more complex and more difficult to specify fully and precisely, and monitors or serialized objects because they seem to interact with the language in more complex ways. However, we recognize that Semaphores are not a fully satisfactory mechanism, and would welcome detailed proposals for better ones.

Execution state

<Context> := <Process> currentContext

Return the current context of a process, if Suspended, or nil, if Dead. Attempting to read the current context of a Waiting or Active process is an error: a performance monitor like the current Smalltalk-80 MessageTally should suspend the process being monitored, read or copy as much of its state as desired, and then resume the process.

<Process> currentContext: <Context>

Set the current context of a Suspended or Dead process. The context must be on the sender chain of the process. If the process was Dead, it becomes Suspended.

<Symbol> := <Process> nextStepAction

Return a symbol indicating what the (Suspended) process will do in response to the next 'step' message: #send, #return, or #advance (execute straight-line code).

<Process> step

Single-step the (Suspended) process to the next send or return (if executing straight-line code) or through the send or return. Note that a send may cause the process to change state from Suspended to Waiting, and a return may cause the process to die (if it returns from its root context).

<Process> restartContext: <Context>

Adjust the state of a Suspended process so that when it next becomes active, it will start executing the method of the indicated context from the beginning again. The context must be on the sender chain of the process.

New scope mechanisms for the Smalltalk-80 language

L. Peter Deutsch
Xerox, ParcPlace Systems

This is a working document proposing mechanisms for more flexible control of name scopes in the Smalltalk-80 language, including private and nested name spaces for classes.

Comments are solicited. For changes in successive versions, consult the version history.

Version history:

[2] 7 May 1987: split off this document (scopes) from pkg87; incorporated comments from Richard Steiger, Mark Miller, and February implementors' workshop (no substantive changes, only clarifications).

[1] 25 February 1987: first version (pkg87), distributed to implementors at the Feb. 26-27 workshop.

Introduction

Conventional language systems support independent development of application packages with encapsulation mechanisms (such as name hiding) that allow them to be combined into a single executable program. The Smalltalk-80 system currently does not provide facilities, which substantially hinders the ability of developers to produce widely-usable packages. This document proposes a more capable name scoping facility for the Smalltalk-80 system. It does not address the other problem of encapsulation, namely how to create truly protected objects using the name scoping facilities in concert with object-oriented design.

In theory, an object-oriented language such as Smalltalk provides excellent modularity, since messages are interpreted relative to the class. In practice, there are several barriers to meeting this promise:

- All class names are global, so any object can send a message to any class. This makes protection awkward, because typical applications have both public and private classes. It also raises a problem of name clashes between classes of different applications.
- The system facilities have been optimized for development environments, where it is very valuable to have all relevant symbolic information readily available (such as message and variable names), as opposed to delivery environments, where it may be more desirable (for reasons of both space and protection) to do without this information.

We propose to address these problems through a generalization of the present class variable / pool dictionary mechanism, which allows the

easy creation of private scopes for static variables and constants (including classes).

Implementing the proposal would involve very minor changes to the Smalltalk compiler, and minor changes to the Debugger and Inspector.

The present proposal is designed primarily for helping to organize large-scale systems composed of many weakly interacting application areas, i.e. it is aimed primarily at the problem of name conflicts.

Scopes

The present Smalltalk-80 language has four kinds of instantiable variables (variables which can exist in multiple instances):

- Block variables (arguments and temporaries);
- Method variables (arguments and temporaries);
- Instance variables;
- Class instance variables.

The present system has three kinds of static variables (variables which only exist in a single instance):

- Class variables;
- Pool variables;
- Global variables (in Smalltalk or Undeclared).

We believe that it is both feasible and desirable to collapse these three kinds of static variables into a single notion of a static variable dictionary: the proposals below indicate how to do this at an implementation level, but do not specify what effect it might have on the language.

In the long term, we believe that it is both feasible and highly desirable to reduce the number of conceptually different kinds of variable, ideally to a single concept of an instantiable variable and instantiable scope. However, at this time we have no proposal for doing this. In particular, we are not prepared to propose eliminating the instantiable / static distinction, even though static variables are conceptually at odds with the notion of instantiable scopes.

Independent of this issue, we observe that in order for the Debugger to evaluate expressions relative to a particular context, it must be able to invoke the compiler and pass it extra information describing how to interpret and resolve names referring to local variables of the context. We also note that the Tektronix Smalltalk system provides a facility for defining static variables local to individual workspaces: this also implies the ability to compile (or at least evaluate) expressions in a specified naming context. We propose here to extend the compiler interface in a way that allows programmers to construct arbitrary scope structures involving static variables. This solves the problem at the implementation level, but not the language level: we do not address the latter problem here.

The key concept in our approach to more flexible name resolutions is to identify an abstract class NameScope and its protocol. In the present version of this proposal, we only deal with static scopes, i.e. those for which supplying only the name is sufficient to obtain a value,

without needing any context information. We hope to extend this to include instantiable scopes in the future.

Conceptually, a NameScope is an ordered collection of dictionaries. Dictionaries earlier in the ordering take precedence over those appearing later, if the same name appears in more than one. (Allowing the same name to appear more than once is important, since otherwise one might have to change names in a local scope as a result of something happening in an outer scope in which one has no interest.) However, a NameScope behaves (through its public protocol) like a single dictionary, i.e. as though it were a dictionary in which the keys are all the names recognized within the scope. For example, an evaluator can use the at: or at:ifAbsent: message to obtain the value of a variable; a compiler can use the associationAt: message to obtain a reference to the binding of a name.

The only major public protocol distinction between NameScopes and Dictionaries is how they are created. In particular, NameScopes are created using the following messages:

<StaticNameScope> := <NameScope class> empty

Returns a NameScope that recognizes no variables.

<StaticNameScope> := <StaticNameScope> precededBy: <Dictionary>

Returns a new NameScope which looks up names in the Dictionary before looking them up in the receiver. In other words, if we think of a NameScope as a sequence of Dictionaries, in which Dictionaries appearing earlier in the list take precedence over those appearing later, this message produces a new NameScope with the Dictionary added at the head of the list. For example, the standard Smalltalk-80 global name scope is produced by ((NameScope empty precededBy: Undeclared) precededBy: Smalltalk). The Dictionaries are incorporated into the NameScope by reference in the following sense:

- If a name is added or removed in a Dictionary, subsequent compilations using any NameScope that includes that Dictionary will take account of the change.
- If a name is added or removed in a Dictionary, already-compiled methods that reference variables in that dictionary MAY OR MAY NOT take account of the change. In other words, we do NOT require that static variables be looked up in a chain of Dictionaries at the moment of use, the way that messages are looked up in MethodDictionaries at the moment of sending.

<InstantiableNameScope> := <Context> nameScope

Returns a NameScope appropriate for evaluating expressions in the given Context.

<ExecutableMethod> := <Behavior> compile: <String> in: <NameScope> ifError: <BlockClosure>

Compiles the String as a method for the given Behavior, using the NameScope as the outer context for names not defined in the Behavior or its superclasses. Note that the NameScope may have been

obtained from a Context: this makes it possible, for example, to compile a new block relative to an existing context. This is also the first step in evaluating expressions relative to a context. We do not specify what happens if a name is not defined in the NameScope: we encourage implementors to provide a "soft" failure alternative such as the Undeclared dictionary provided in the present Smalltalk-80 system.

Using this mechanism, an application can construct classes whose names are entirely local to that application. Obviously one wants to have a linguistic mechanism for this, not just a set of procedural or even interactive tools. This is partly for user convenience, but also because there is a protection issue here: the messages for constructing and accessing scopes, and compiling, must be protected from unauthorized use. This question of so-called "meta-level" access is beyond the scope of this document.