
Self Handbook Documentation

Release for Self 2017.1

Russell Allen (Ed.)

May 05, 2017

CONTENTS

1	Introduction	3
1.1	History	3
1.2	System Overview	3
2	Getting Started	5
2.1	Downloading Prebuilt VM and snapshot	5
2.2	Running on OS X	5
2.3	Running on Linux	5
2.4	Building your own system	5
3	Language Reference	7
3.1	Objects	7
3.2	Slot descriptors	11
3.3	Expressions	16
3.4	Lexical elements	22
4	The Self World	27
4.1	World Organization	27
4.2	The Roots of Behavior	28
4.3	Blocks, Booleans, and Control Structures	29
4.4	Numbers and Time	31
4.5	Collections	32
4.6	Pairs	35
4.7	Mirrors	36
4.8	Messages	36
4.9	Processes and the Prompt	37
4.10	Foreign Objects	37
4.11	I/O and Unix	38
4.12	Other Objects	39
4.13	How to build the world	39
4.14	How to use the low-level interrupt facilities	41
4.15	Using the textual debugger	42
4.16	Logging	42
5	A Guide to Programming Style	45
5.1	Behaviorism versus Reflection	45
5.2	Objects Have Many Roles	46
5.3	Naming and Printing	47
5.4	How to Return Multiple Values	48
5.5	Substituting Values for Blocks	49

5.6	<code>nil</code> Considered Naughty	49
5.7	Hash and =	49
5.8	Equality, Identity, and Indistinguishability	50
6	How to Program in Self	51
6.1	Introduction	51
6.2	Browsing Concepts	51
6.3	Hacking Objects	68
6.4	The Transporter	71
7	Morphic: The Self User Interface Framework	77
7.1	Overview	77
7.2	Composite Morphs	78
7.3	Morph Traits and Prototypes	79
7.4	Saving a Composite Morph	90
7.5	Handling User Input	90
7.6	Drag and Drop	92
7.7	Automatic Layout	92
7.8	Animation	94
7.9	Other Issues	96
7.10	Morph Responsibilities	99
7.11	Some Useful Morphs	99
7.12	The Graphical Environment	100
8	Virtual Machine Reference	103
8.1	Building a VM	103
8.2	Startup options	104
8.3	System-triggered messages	104
8.4	Run-time message lookup errors	104
8.5	Low-level error messages	106
8.6	An example	106
8.7	Lookup errors	107
8.8	Programmer defined errors	107
8.9	Primitive errors	107
8.10	Nonrecoverable process errors	108
8.11	Fatal errors	108
8.12	The initial Self world	109
8.13	Option Primitives	113
8.14	Interfacing with other languages	114
9	References	131
10	Appendices	133
10.1	Glossary	133
10.2	Lexical overview	135
10.3	Syntax overview	136
10.4	Built-in types	136
10.5	Useful Selectors	136
10.6	Every Menu Item in the Programming Environment	144
10.7	The system monitor	146
10.8	Primitives	149
11	Extras	151
11.1	The Original Self UI	151

Release for Self 2017.1

Date May 05, 2017

Edited by Russell Allen.

Authors (in alphabetical order): Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Tobias Pape, Randall B. Smith, David Ungar and Mario Wolczko.

Thanks to Ganesh R for his transcription services.

INTRODUCTION

Self is a prototype-based dynamic object-oriented programming language, environment, and virtual machine centered around the principles of simplicity, uniformity, concreteness, and liveness.

Self includes a programming language, a collection of objects defined in the Self language, and a programming environment built in Self for writing Self programs. The language and environment attempt to present objects to the programmer and user in as direct and physical a way as possible. The system uses the prototype-based style of object construction.

Self is available for Solaris, Linux and natively on MacOS X under a BSD-like licence; we would be very interested in anyone prepared to make a Windows port.

1.1 History

The first version of the Self language was designed in 1986 by David Ungar and Randall B. Smith at Xerox PARC. A series of Self implementations and a graphical programming environment were built at Stanford University by Craig Chambers, Urs Hölzle, Ole Agesen, Elgin Lee, Bay-Wei Chang, and David Ungar. The project continued at Sun Microsystems Laboratories, where it benefited from the efforts of Randall B. Smith, Mario Wolczko, John Maloney, and Lars Bak. Smith and Ungar jointly led it there. Work on the project officially ceased in 1995

Release 4.0 contained an entirely new user interface and programming environment designed for “serious” programming, enabling the programmer to create and modify objects entirely within the environment, and then save the object into files for distribution purposes. The metaphor used to present an object to the user is that of an outliner, allowing the user to view varying levels of detail. Also included in the environment is a graphical debugger, and tools for navigation through the system.

Release 4.4 was the first release for Linux x86.

1.2 System Overview

This section contains an overview of the system and its implementation; it can be skipped if you wish to get started as quickly as possible.

Although Self runs as a single UNIX¹ process, or a single Macintosh application, it really has two parts: the *virtual machine* (VM) and the *Self world*, the collection of Self objects that are the Self prototypes and programs:

The VM executes Self programs specified by objects in the Self world and provides a set of *primitives* (which are methods written in C++) that can be invoked by Self methods to carry out basic operations like integer arithmetic, object copying, and I/O. The Self world distributed with the VM is a collection of Self objects implementing various

¹ UNIX is a trademark of AT&T Bell Laboratories.

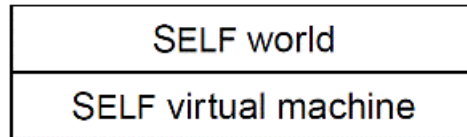


Fig. 1.1: The Self system

traits and *prototypes* like cloning traits and dictionaries. These objects can be used (or changed) to implement your own programs.

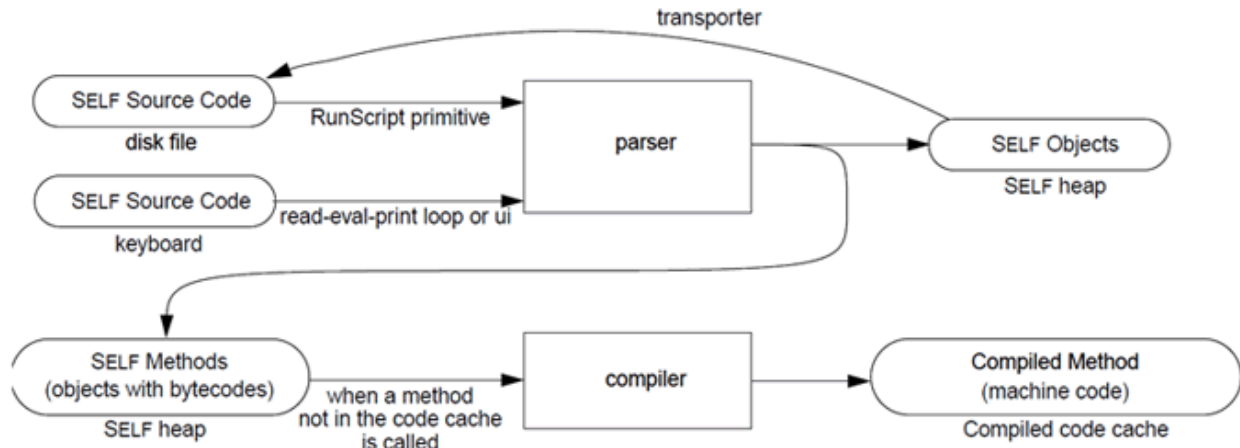


Fig. 1.2: How Self programs are compiled.

Self programs are translated to machine code in a two-stage process (see Fig. 1.2). Code typed in at the prompt, through the user interface, or read in from a file is parsed into Self objects. Some of these objects are data objects; others are methods. Methods have their own behavior which they represent with *bytecodes*. The bytecodes are the instructions for a very simple virtual processor that understands instructions like “push receiver” or “send the ‘x’ message.” In fact, Self bytecodes correspond much more closely to source code than, say, Smalltalk-80 bytecodes. (See [CUL89] for a list of the Self byte codes.) The *raison d’être* of the virtual machine is to pretend that these bytecodes are directly executed by the computer; the programmer can explore the Self world down to the bytecode level, but no further. This pretense ensures that the behavior of a Self program can be understood by looking only at the Self source code.

The second stage of translation is the actual *compilation* of the bytecodes to machine code. This is how the “execution” of bytecodes is implemented—it is totally invisible on the Self level except for side effects like execution speed and memory usage. The compilation takes place the first time a message is actually sent; thus, the first execution of a program will be slower than subsequent executions.

Actually, this explanation is not entirely accurate: the compiled method is specialized on the type of the receiver. If the same message is later sent to a receiver of different type (e.g., a float instead of an integer), a new compilation takes place. This technique is called *customization*; see [CU89] for details. Also, the compiled methods are placed into a cache from which they can be flushed for various reasons; therefore, they might be recompiled from time to time. Furthermore, the current version of the compiler will recompile and reoptimize frequently used code, using information gathered at run-time as to how the code is being used; see [HCU91] for details.

Don’t be misled by the term “compiled method” if you are familiar with Smalltalk: in Smalltalk terminology it denotes a method in its bytecode form, but in Self it denotes the native machine code form. In Smalltalk there is only one compiled method per source method, but in Self there may be several different compiled methods for the same source method (because of customization).

GETTING STARTED

Last updated 5 January 2014 for Self 4.5.0

2.1 Downloading Prebuilt VM and snapshot

The easiest way to get started with Self is to download a prebuilt VM and a snapshot of the default Self world from the Self website.

For OS X, this means a DMG file which will mount when double clicked. On Linux this means a `.tar.gz` file.

2.2 Running on OS X

Once the DMG has been mounted, find it in Finder and drag the `'Self Control.app'` to your Applications folder, and the `'Clean.snap'` file somewhere convenient. This file is a snapshot of the default Self world.

2.3 Running on Linux

Untar the downloaded `.tar.gz` file:

```
tar zxvf Self.tar.gz
```

This should give you at least two files, a binary `'Self'` which is the VM, and the file `'Clean.snap'` which is a snapshot of the default Self world. Make sure the VM is executable:

```
chmod u+x Self
```

and run the VM, passing the snapshot as an argument:

```
Self -s Clean.self
```

2.4 Building your own system

It is relatively easy to build the two key components of Self, the Self World and the Self Virtual Machine. They both start from downloading the Self sources from the GitHub repository by the following git command:

```
git clone https://github.com/russellallen/self/
```

and then follow chapters *8.1 Instructions on how to build the VM* and *4.13 Instructions on how to build a Self World*.

LANGUAGE REFERENCE

This chapter specifies Self's syntax and semantics. An early version of the syntax was presented in the original Self paper by Ungar and Smith [US87]; this chapter incorporates subsequent changes to the language. The presentation assumes a basic understanding of object-oriented concepts.

The syntax is described using Extended Backus-Naur Form (EBNF). Terminal symbols appear in Courier and are enclosed in single quotes; they should appear in code as written (not including the single quotes). Non-terminal symbols are italicized. The following table describes the metasymbols:

META-SYMBOL	FUNCTION	DESCRIPTION
(and)	grouping	used to group syntactic constructions
[and]	option	encloses an optional construction
{ and }	repetition	encloses a construction that may be repeated zero or more times
\	alternative	separates alternative constructions
→	production	separates the left and right hand sides of a production

A glossary of terms used in this document can be found in [Appendix A](#).

3.1 Objects

Objects are the fundamental entities in Self; every entity in a Self program is represented by one or more objects. Even control is handled by objects: *blocks* (§3.1.7) are Self closures used to implement user-defined control structures. An object is composed of a (possibly empty) set of *slots* and, optionally, *code* (§3.1.5). A slot is a name-value pair; slots contain references to other objects. When a slot is found during a *message lookup* (§3.3.6) the object in the slot is evaluated.

Although everything is an object in Self, not all objects serve the same purpose; certain kinds of objects occur frequently enough in specialized roles to merit distinct terminology and syntax. This chapter introduces two kinds of objects, namely data objects (“plain” objects) and the two kinds of objects with code, ordinary methods and block methods.

3.1.1 Syntax

Object literals are delimited by parentheses. Within the parentheses, an object description consists of a list of slots delimited by vertical bars (‘|’), followed by the code to be executed when the object is evaluated. For example:

```
( | slot1. slot2 | 'here is some code' printLine )
```

Both the slot list and code are optional: ‘(| |)’ and ‘()’ each denote an empty object¹.

¹ If you wish to use the empty vertical bar notation to create an empty object, note that the parser currently requires a space between the vertical bars.

Block objects are written like other objects, except that square brackets (‘[’ and ‘]’) are used in place of parentheses:

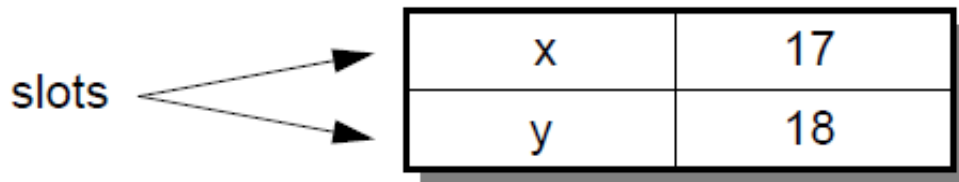
```
[ | slot1. slot2 | 'here is some code in a block' printLine ]
```

A *slot list* consists of a (possibly empty) sequence of *slot descriptors* (§3.2) separated by periods. A period at the end of the slot list is optional².

The code for an object is a sequence of *expressions* (§3.3) separated by periods. A trailing period is optional. Each expression consists of a series of *message sends* and *literals*. The last expression in the code for an object may be preceded by the ‘^’ operator (§3.1.8).

3.1.2 Data objects

Data objects are objects without code. Data objects can have any number of slots. For example, the object `()` has no slots (i.e., it’s empty) while the object `(| x = 17. y = 18 |)` has two slots, `x` and `y`.



A data object returns itself when evaluated.

3.1.3 The assignment primitive

A slot containing the assignment primitive is called an *assignment slot* (§3.2.2). When an assignment slot is evaluated, the argument to the message is stored in the corresponding *data slot* (§3.2) in the same object (the slot whose name is the assignment slot’s name minus the trailing colon), and the *receiver* (§3.3.4) is returned as the result. (Note: this means that the value of an assignment statement is the left-hand side of the assignment statement, not the right-hand side as it is in Smalltalk, C, and many other languages. This is a potential source of confusion for new Self programmers.)

3.1.4 Objects with code

The feature that distinguishes a *method object* from a data object is that it has *code*, whereas a data object does not. Evaluating a method object does not simply return the object itself, as with simple data objects; rather, its code is executed and the resulting value is returned.

3.1.5 Code

Code is a sequence of *expressions* (§3.3). These expressions are evaluated in order, and the resulting values are discarded except for that of the final expression, whose value determines the result of evaluating the code.

² But in that case make sure you put a space after the period, otherwise you will get an obscure error message from the parser.

The actual arguments in a message send are evaluated from left to right before the message is sent. For instance, in the expression:

```
1 to: 5 * i By: 2 * j Do: [| :k | k print ]
```

1 is evaluated first, then $5 * i$, then $2 * j$, and then `[| :k | k print]`. Finally, the `to:By:Do:` message is sent. The associativity and precedence of messages is discussed in section 3.3.

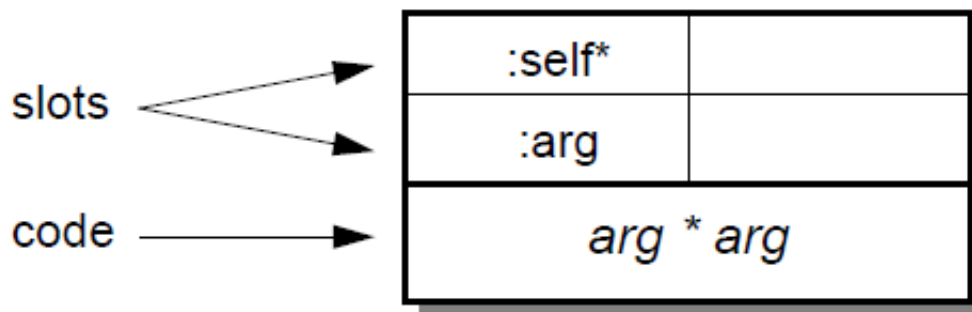
3.1.6 Methods

Ordinary methods (or simply “methods”) are methods that are not embedded in other code. A method can have *argument slots* (§3.2.3) and/or local slots. An ordinary method always has an implicit *parent* (§3.2.4) argument slot named `self`. Ordinary methods are Self’s equivalent of Smalltalk’s methods.

If a slot contains a method, the following steps are performed when the slot is evaluated as the result of a message send:

- The method object is *cloned*, creating a new *method activation object* containing slots for the method’s arguments and locals.
- The clone’s `self` parent slot is initialized to the receiver of the message.
- The clone’s argument slots, if any, are initialized to the values of the corresponding actual arguments.
- The code of the method is executed in the context of this new activation object.

For example, consider the method `(| :arg | arg * arg)`:

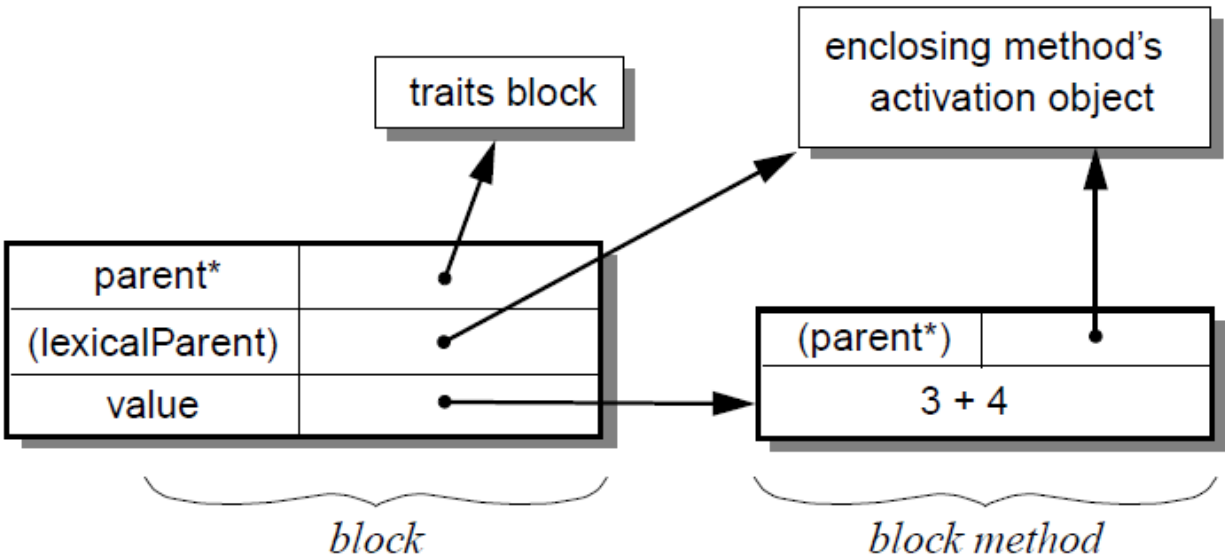


This method has an argument slot `arg` and returns the square of its argument.

3.1.7 Blocks

Blocks are Self closures; they are used to implement user-defined control structures. A block literal (delimited by square brackets) defines two objects: the *block method object*, containing the block’s code, and an enclosing *block data object*. The block data object contains a parent pointer (pointing to the object containing the shared behavior for block objects) and a slot containing the block method object. Unlike an ordinary method object, the block method object does not contain a `self` slot. Instead, it has an anonymous parent slot that is initialized to point to the activation object for the lexically enclosing block or method. As a result, *implicit-receiver messages* (§3.3.4) sent within a block method are lexically scoped. The block method object’s anonymous parent slot is invisible at the Self level and cannot be accessed explicitly.

For example, the block `[3 + 4]` looks like³:



The block method's selector is based on the number of arguments. If the block takes no arguments, the selector is `value`. If it takes one argument, the selector is `value:`. If it takes two arguments, the selector is `value:With:`, for three the selector is `value:With:With:`, and for more the selector is just extended by enough `With:`'s to match the number of block arguments.

Block evaluation has two phases. In the first phase, a block object is created because the block is evaluated (e.g., it is used as an argument to a message send). The block is cloned and given a pointer to the activation record for its lexically enclosing scope, the current activation record. In the second phase, the block's method is evaluated as a result of sending the block the appropriate variant of the `value` message. The block method is then cloned, the argument slots of the clone are filled in, the anonymous parent slot of the clone is initialized using the scope pointer determined in phase one, and, finally, the block's code is executed.

It is an error to evaluate a block method after the activation record for its lexically enclosing scope has returned. Such a block is called a *non-lifo* block because returning from it would violate the last-in, first-out semantics of activation object invocation.

This restriction is made primarily to allow activation records to be allocated from a stack. A future release of Self may relax this restriction, at least for blocks that do not access variables in enclosing scopes.

3.1.8 Returns

A *return* is denoted by preceding an expression by the `^` operator. A return causes the value of the given expression to be returned as the result of evaluating the method or block. Only the last expression in an object may be a return.

The presence or absence of the `^` operator does not effect the behavior of ordinary methods, since an ordinary method always returns the value of its final expression anyway. In a block, however, a return causes control to be returned from the ordinary method containing that block, immediately terminating that method's activation, the block's activation, and all activations in between. Such a return is called a *non-local return*, since it may "return through" a number of activations. The result of the ordinary method's evaluation is the value returned by the *non-local return*. For example, in the following method:

³ All block objects have the same parent, an object containing the shared behavior for blocks

```
assertPositive: x = (
  x > 0 ifTrue: [ ^ 'ok' ].
  error: 'non-positive x' )
```

the `error:` message will not be sent if `x` is positive because the non-local return of `'ok'` causes the `assertPositive:` method to return immediately.

3.1.9 Construction of object literals

Object literals are constructed during parsing — the parser converts objects in textual form into real Self objects. An object literal is constructed as follows:

- First, the slot initializers of every slot are evaluated from left to right. If a slot initializer contains another object literal, this literal is constructed before the initializer containing it is evaluated. If the initializer is an expression, it is evaluated in the context of the lobby.
- Second, the object is created, and its slots are initialized with the results of the evaluations performed in the first step.

Slot initializers are *not* evaluated in the lexical context, since none exists at parse time; they are evaluated in the context of an object known as the `lobby`. That is, the initializers are evaluated as if they were the code of a method in a slot of the `lobby`. This two-phase object construction process implies that slot initializers may not refer to any other slots within the constructed object (as with Scheme's `let*` and `letrec` forms) and, more generally, that a slot initializer may not refer to any textually enclosing object literal.

3.2 Slot descriptors

An object can have any number of slots. Slots can contain data (*data slots*) or methods. Some slots have special roles: *argument slots* are filled in with the actual arguments during a message (§3.3.7), and *parent slots* specify inheritance relationships (§3.3.8).

A *slot descriptor* consists of an optional privacy specification, followed by the slot name and an optional initializer.

3.2.1 Read-only slots

A slot name followed by an equals sign (`'='`) and an expression represents a *read-only slot* initialized to the result of evaluating the expression in the root context.

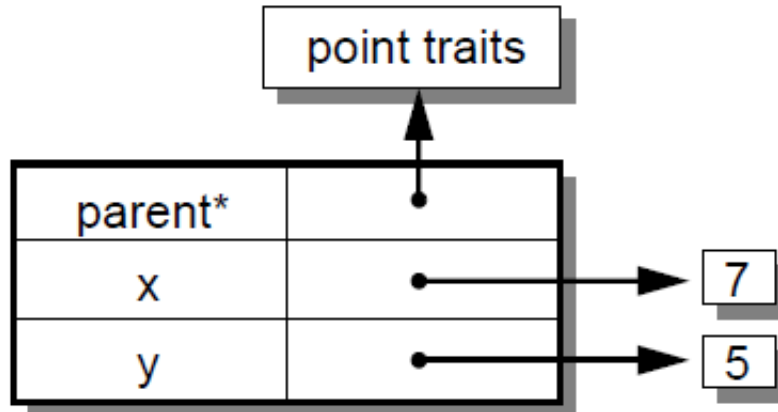
For example, a constant point might be defined as:

```
( | parent* = traits point.
  x = 3 + 4.
  y = 5.
| )
```

The resulting point contains three initialized read-only slots:

3.2.2 Read/write slots

There is no separate assignment operation in Self. Instead, assignments to data slots are message sends that invoke the assignment primitive. For example, a data slot `x` is assignable if and only if there is a slot in the same object with the same name appended with a colon (in this case, `x:`), containing the assignment primitive. Therefore, assigning `17` to



slot `x` consists of sending the message `x: 17`. Since this is indistinguishable from a message send that invokes a method, clients do not need to know if `x` and `x:` comprise data slot accesses or method invocations.

An identifier followed by a left arrow (the characters ‘<’ and ‘-’ concatenated to form ‘<-’) and an expression represents an initialized *read/write variable* (assignable data slot). The object will contain both a data slot of that name and a corresponding assignment slot whose name is obtained by appending a colon to the data slot name. The initializing expression is evaluated in the root context and the result stored into the data slot at parse time.

For example, an initialized mutable point might be defined as:

```
( | parent* = traits point.
  x <- 3 + 4.
  y <- 5.
 | )
```

producing an object with two data slots (`x` and `y`) and two assignment slots (`x:` and `y:`) containing the assignment primitive (depicted with \leftarrow)⁴:

An identifier by itself specifies an assignable data slot initialized to *nil*⁵. Thus, the slot declaration `x` is a shorthand notation for `x <- nil`.

For example, a simple mutable point might be defined as:

```
( | x. y. | )
```

producing:

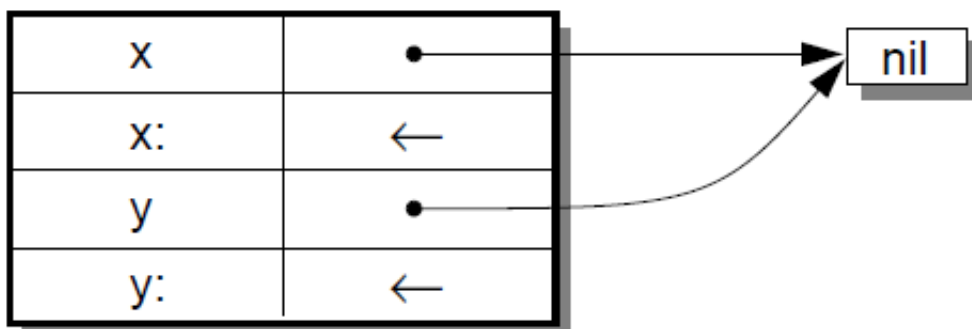
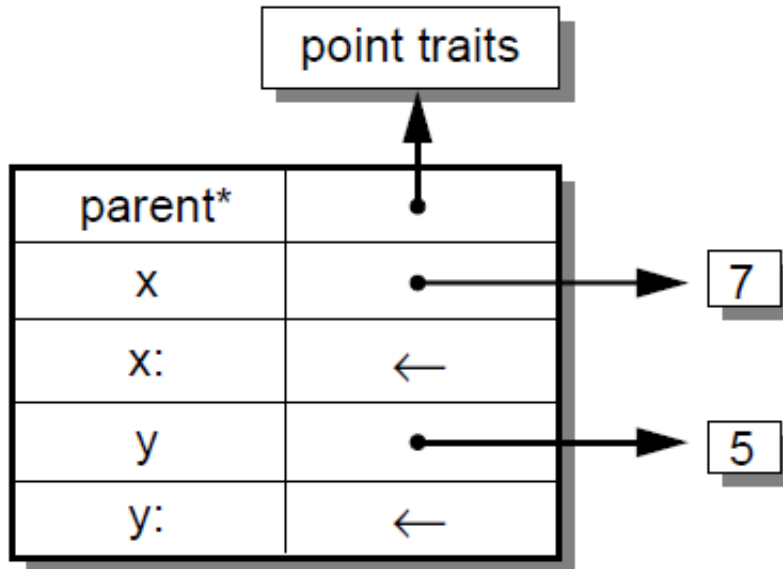
3.2.3 Slots containing methods

If the initializing expression is an object literal with code, that object is stored into the slot *without evaluating the code*. This allows a slot to be initialized to a method by storing the method itself, rather than its result, in the slot⁶. Methods may only be stored in read-only slots. A method automatically receives a parent argument slot named `self`. For example, a point addition method can be written as:

⁴ In the user interface a read/write slot is depicted as a single slot with a colon labelling the button used to access the value of the slot; the assignment slot is not shown, to save screen space. In contrast, a read-only slot has an equals sign on the button.

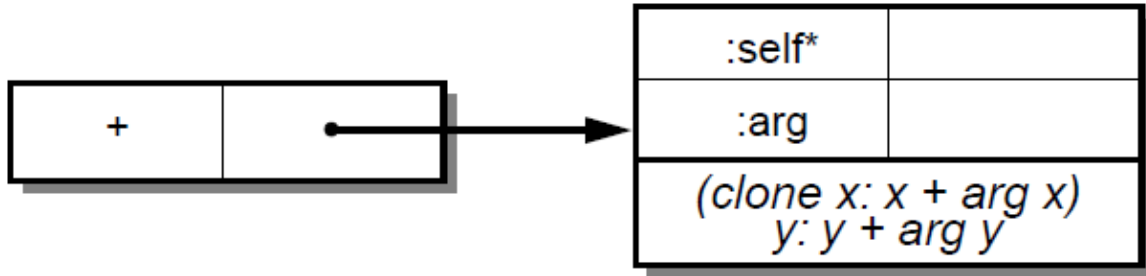
⁵ `nil` is a predefined object provided by the implementation. It is intended to indicate “not a useful object.”

⁶ Although a block may be assigned to a slot at any time, it is often not useful to do so: evaluating the slot may result in an error because the activation record for the block’s lexically enclosing scope will have returned; see §3.1.7.



```
( |
  + = ( | :arg | (clone x: x + arg x) y: y + arg y ).
| )
```

producing:



A slot name beginning with a colon indicates an *argument* slot. The prefixed colon is not part of the slot name and is ignored when matching the name against a message. Argument slots are always read-only, and no initializer may be specified for them. As a syntactic convenience, the argument name may also be written immediately after the slot name (without the prefixed colon), thereby implicitly declaring the argument slot. Thus, the following yields exactly the same object as above:

```
( |
  + arg = ( (clone x: x + arg x) y: y + arg y ).
| )
```

The + slot above is a *binary slot* (§3.3.2), taking one argument and having a name that consists of operator symbols. Slots like x or y in a point object are *unary slots* (§3.3.1), which take no arguments and have simple identifiers for names. In addition, there are *keyword slots* (§3.3.3), which handle messages that require one or more arguments. A keyword slot name is a sequence of identifiers, each followed by a colon.

The arguments in keyword methods are handled analogously to those in binary methods: each colon-terminated identifier in a keyword slot name requires a corresponding argument slot in the keyword method object, and the argument slots may be specified either all in the method or all interspersed with the selector parts.

For example:

```
( |
  ifTrue: False: = ( | :trueBlock. :falseBlock |
    trueBlock value ).
| )
```

and

```
( |
  ifTrue: trueBlock False: falseBlock =
    ( trueBlock value ).
| )
```

produce identical objects.

3.2.4 Parent slots

A unary slot name followed by an asterisk denotes a *parent slot*. The trailing asterisk is not part of the slot name and is ignored when matching the name against a message. Except for their special meaning during the message lookup process (§3.3.8), parent slots are exactly like normal unary slots; in particular, they may be assignable, allowing *dynamic inheritance*. Argument slots cannot be parent slots.

3.2.5 Annotations

In order to provide extra information for the programming environment, Self supports annotations on either whole objects or individual slots. Although any object can be an annotation, the Self syntax only supports the textual definition of string annotations. In order to annotate an object, use this syntax:

```
( | {} = 'this object has one slot' snort = 17. | ) }
```

In order to annotate a group of slots, surround them with braces and insert the annotation after the opening brace:

```
( |
  { 'Category: accessing'
    getOne = (...).
    getAnother = (...).
  }
  anUnannotatedSlot.
| )
```

Annotations may nest; if so the Virtual Machine concatenates the annotations strings and inserts a separator character (16r7f)⁷.

⁷ The current programming environment expects a slot annotation to start with one of a number of keywords, including `Category:`, `Comment:`, and `ModuleInfo:`. See the programming environment manual for more details.

3.3 Expressions

Expressions in Self are *messages* sent to some object, the *receiver*. Self message syntax is similar to Smalltalk's. Self provides three basic kinds of messages: unary messages, binary messages, and keyword messages. Each has its own syntax, associativity, and precedence. Each type of message can be sent either to an explicit or implicit receiver.

Productions⁸:

expression	→	constant unary-message binary-message keyword-message '(' expression ')'
constant	→	self number string object
unary-message	→	receiver unary-send resend '.' unary-send
unary-send	→	identifier
binary-message	→	receiver binary-send resend '.' binary-send
binary-send	→	operator expression
keyword-message	→	receiver keyword-send resend '.' keyword-send
keyword-send	→	small-keyword expression { cap-keyword expression }
receiver	→	[expression]
resend	→	resend identifier

The table below summarizes Self's message syntax rules:

MESSAGE	ARGUMENTS	PRECEDENCE	ASSOCIATIVITY	SYNTAX
Unary	0	highest	none	[receiver] identifier
Binary	1	medium	none or left-to-right ¹⁴	[receiver] operator expression
Keyword	>= 1	lowest	right-to-left	[receiver] small-keyword expression { cap-keyword expression }

Parentheses can be used to explicitly specify order of evaluation.

3.3.1 Unary messages

A *unary message* does not specify any arguments. It is written as an identifier following the receiver.

Examples of unary messages sent to explicit receivers:

```
17 print
5 factorial
```

Associativity. Unary messages compose from left to right. An expression to print 5 factorial, for example, is written:

```
5 factorial print
```

and interpreted as:

```
(5 factorial) print
```

Precedence. Unary messages have higher precedence than binary messages and keyword messages.

3.3.2 Binary messages

A *binary message* has a receiver and a single argument, separated by a binary operator. Examples of binary messages:

⁸ In order to simplify the presentation, this grammar is ambiguous; precedence and associativity rules are used to resolve the ambiguities.

¹⁴ Heterogeneous binary messages have no associativity; homogeneous binary messages associate left-to-right.

```
3 + 4
7 <-> 8
```

Associativity. Binary messages have no associativity, except between identical operators (which associate from left to right). For example,

```
3 + 4 + 7
```

is interpreted as

```
(3 + 4) + 7
```

But

```
3 + 4 * 7
```

is illegal: the associativity must be made explicit by writing either

```
(3 + 4) * 7 or 3 + (4 * 7) .
```

Precedence. The precedence of binary messages is lower than unary messages but higher than keyword messages. All binary messages have the same precedence. For example,

```
3 factorial + pi sine
```

is interpreted as

```
(3 factorial) + (pi sine)
```

3.3.3 Keyword messages

A *keyword message* has a receiver and one or more arguments. It is written as a receiver followed by a sequence of one or more keyword-argument pairs. The first keyword must begin with a lower case letter or underscore ('_'); subsequent keywords must be capitalized. An initial underscore denotes that the operation is a *primitive*. A keyword message consists of the longest possible sequence of such keyword-argument pairs; the message selector is the concatenation of the keywords forming the message. Message selectors beginning with an underscore are reserved for *primitives* (10.8).

Example:

```
5 min: 4 Max: 7
```

is the single message `min:Max:` sent to 5 with arguments 4 and 7, whereas

```
5 min: 4 max: 7
```

involves two messages: first the message `max:` sent to 4 and taking 7 as its argument, and then the message `min:` sent to 5, taking the result of `(4 max: 7)` as its argument.

Associativity. Keyword messages associate from right to left, so

```
5 min: 6 min: 7 Max: 8 Max: 9 min: 10 Max: 11
```

is interpreted as

```
5 min: (6 min: 7 Max: 8 Max: (9 min: 10 Max: 11))
```

The association order and capitalization requirements are intended to reduce the number of parentheses necessary in Self code. For example, taking the minimum of two slots *m* and *n* and storing the result into a data slot *i* may be written as:

```
i: m min: n
```

Precedence. Keyword messages have the lowest precedence. For example,

```
i: 5 factorial + pi sine
```

is interpreted as

```
i: ((5 factorial) + (pi sine))
```

3.3.4 Implicit-receiver messages

Unary, binary, and keyword messages are frequently written without an explicit receiver. Such messages use the current receiver (*self*) as the implied receiver. The method lookup, however, begins at the current activation object rather than the current receiver (see §3.1.6 for details on activation objects). Thus, a message sent explicitly to *self* is *not* equivalent to an implicit-receiver send because the former won't search local slots before searching the receiver. Explicitly sending messages to *self* is considered bad style.

Examples:

```
factorial      (implicit-receiver unary message)
+ 3           (implicit-receiver binary message)
max: 5        (implicit-receiver keyword message)
1 + power: 3  (parsed as 1 + (power: 3))
```

Accesses to slots of the receiver (local or inherited) are also achieved by implicit message sends to *self*. For an assignable data slot named *t*, the message *t* returns the contents, and *t: 17* puts 17 into the slot.

3.3.5 Resending messages

A *resend* allows an overriding method to invoke the overridden method. Directed resends allow ambiguities among overridden methods to be resolved by constraining the lookup to search a single parent slot. Both resends and directed resends may change the name of the message being sent from the name of the current method, and may pass different arguments than the arguments passed to the current method. The receiver of a resend or a directed resend must be the implicit receiver.

Intuitively, resend is similar to Smalltalk's *super send* and CLOS' *call-next-method*.

A resend is written as an implicit-receiver message with the reserved word *resend*, a period, and the message name. No whitespace may separate *resend*, the period, and the message name.

Examples:

```
resend.display
resend.+ 5
resend.min: 17 Max: 23
```

A *directed resend* constrains the resend through a specified parent. It is written similar to a normal resend, but replaces *resend* with the name of the parent slot through which the resend is directed.

Examples:

```
listParent.height
intParent.min: 17 Max: 23
```

Only implicit-receiver messages may be delegated via a resend or a directed resend⁹.

3.3.6 Message lookup semantics

This section describes the semantics of message lookups in Self. In addition to an informal textual description, the lookup semantics are presented in pseudo-code using the following notation:

s.name	The name of slot s.
s.contents	The object contained in slot s.
s.isParent	True iff s is a parent slot.
{s ∈ obj pred(s)}	The set of all slots of object obj that satisfy predicate pred.
S	The cardinality of set S.

The message sending semantics are decomposed into the following functions:

send(rec, sel, args)	The message send function (§3.3.7).
lookup(obj, rec, sel, V)	The lookup algorithm (§3.3.8).
undirected_resend(...)	The undirected message resend function (§3.3.9).
directed_resend(...)	The directed message resend function (§3.3.10).
eval(rec, M, args)	The slot evaluation function as described informally throughout §3.1.5.

3.3.7 Message send

There are two kinds of message sends: a *primitive send* has a selector beginning with an underscore (‘_’) and calls the corresponding primitive operation. Primitives are predefined functions provided by the implementation. A *normal send* does a lookup to obtain the target slot; if the lookup was successful, the slot is subsequently evaluated. If the slot contains a data object, then the data object is simply returned. If the slot contains the assignment primitive, the argument of the message is stored in the corresponding data slot. Finally, if the slot contains a method, an activation is created and run as described in §3.1.6.

If the lookup fails, the lookup error is handled in an implementation-defined manner; typically, a message indicating the type of error is sent to the object which could not handle the message.

The function *send(rec, sel, args)* is defined as follows:

Input:

rec, the receiver of the message
 sel, the message selector
 args, the actual arguments

Output:

res, the result object

Algorithm

```
if begins_with_underscore(sel)
then invoke_primitive(rec, sel, args)           "primitive call"
else M ← lookup(rec, sel, ∅)                   "do the lookup"
case
```

⁹ General delegation for explicit receiver messages is supported through primitives in the implementation (see Appendix 9.8).

```

    | M | = 0: error: message not understood
    | M | = 1: res ← eval(rec, M, args)           "see §2.1"
    | M | > 1: error: ambiguous message send
  end
end
return res

```

3.3.8 The lookup algorithm

The lookup algorithm recursively traverses the inheritance graph, which can be an arbitrary graph (including cyclic graphs). No object is searched twice along any single path. The search begins in the object itself and then continues to search every parent. Parent slots are not evaluated during the lookup. That is, if a parent slot contains an object with code, the code will not be executed; the object will merely be searched for matching slots.

The function *lookup(obj, sel, V)* is defined as follows:

Input:

obj, the object being searched for matching slots
sel, the message selector
V, the set of objects already visited along this path

Output:

M, the set of matching slots

Algorithm:

```

if obj ∈ V
then M ← ∅                               "cycle detection"
else M ← {s ∈ obj | s.name = sel}         "try local slots"
    if M = ∅ then M ← parent_lookup(obj, sel, V) end "try parent slots"
end
return M

```

Where *parent_lookup(obj, sel, V)* is defined as follows:

```

P ← {s ∈ obj | s.isParent}               "all parents"
M ← v lookup(s.contents, sel, V v {obj})  "recursively search parents"
    s ∈ P
return M

```

3.3.9 Undirected Resend

An undirected resend ignores the sending method holder (the object containing the currently running method) and continues with its parents.

The function *undirected_resend(rec, smh, sel, args)* is defined as follows:

Input:

rec, the receiver of the message
smh, the sending method holder
sel, the message selector
args, the actual arguments

Output:

res, the result object

Algorithm:

```
M ← parent_lookup(smh, sel, ∅)           "do the lookup"
case
  | M | = 0: error: message not understood
  | M | = 1: res ← eval(rec, M, args)     "see §2.1"
  | M | > 1: error: ambiguous message send
end
return res
```

3.3.10 Directed Resend

A directed resend looks only in one slot in the sending method holder.

The function *directed_resend*(*rec*, *smh*, *del*, *sel*, *args*) is defined as follows:

Input:

rec, the receiver of the message
 smh, the sending method holder
 del, the name of the delegatee
 sel, the message selector
 args, the actual arguments

Output:

res, the result object

Algorithm:

```
D ← {s ∈ smh | s.name = del}           "find delegatee"
if | D | = 0 then error: missing delegatee "one or none"
M ← lookup(smh.del, sel, ∅)           "do the lookup"
case
  | M | = 0: error: message not understood
  | M | = 1: res ← eval(rec, M, args)     "see §2.1"
  | M | > 1: error: ambiguous message send
end
return res
```

3.4 Lexical elements

This chapter describes the lexical structure of Self programs — how sequences of characters in Self source code are grouped into lexical tokens. In contrast to syntactic elements described by productions in the rest of this document, the elements of lexical EBNF productions may not be separated by whitespace, i.e. there may not be whitespace within a lexical token. Tokens are formed from the longest sequence of characters possible. Whitespace may separate any two tokens and must separate tokens that would be treated as one token otherwise.

3.4.1 Character set

Self programs are written using the following characters:

- *Letters*. The fifty-two upper and lower case letters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

- *Digits*. The ten numeric digits:

```
0123456789
```

- *Whitespace*. The formatting characters: space, horizontal tab (ASCII HT), newline (NL), carriage return (CR), vertical tab (VT), backspace (BS), and form feed (FF). (Comments are also treated as whitespace.)
- *Graphic characters*. The 32 non-alphanumeric characters:

```
!@#$%^&*()_+|=|\~`{ } [ ] ; ' " < > , . ? /
```

3.4.2 Identifiers

An *identifier* is a sequence of letters, digits, and underscores ('_') beginning with a lowercase letter or an underscore. Case is significant: `apoint` is not the same as `aPoint`.

Productions:

small-letter	→	'a' 'b' ... 'z'
cap-letter	→	'A' 'B' ... 'Z'
letter	→	small-letter cap-letter
identifier	→	(small-letter '_') {letter digit '_'}

Examples:

```
i _IntAdd cloud9 m a_point
```

The two identifiers `self` and `resend` are reserved. Identifiers beginning with underscores are reserved for primitives.

3.4.3 Keywords

Keywords are used as slot names and as message names. They consist of an identifier or a capitalized identifier followed by a colon (':').

Productions:

small-keyword	→	identifier ':'
cap-keyword	→	cap-letter {letter digit '_'} ':'

Examples:

```
at: Put: _IntAdd:
```

3.4.4 Arguments

A colon followed by an identifier denotes an *argument* slot name.

Productions:

arg-name	→	':' identifier
----------	---	----------------

Example:

```
:name
```

3.4.5 Operators

An *operator* consists of a sequence of one or more of the following characters:

```
! @ # $ % ^ & * - + = ~ / ? < > , ; | ' \
```

Two sequences are reserved and are not operators:

```
| ^
```

Productions:

op-char	→	'!' '@' '#' '\$' '%' '^' '&' '*' '-' '+' '=' '~' '/' '?' '<' '>' ',' ';' ' ' "'" "\"
operator	→	op-char {op-char}

Examples:

```
+ - && \|\| <-> % # @ ^
```

3.4.6 Numbers

Integer literals are written as a sequence of digits, optionally prefixed with a minus sign and/or a base¹⁰. No whitespace is allowed between a minus sign and the digit sequence¹¹. Real constants may be either written in fixed-point or exponential form.

Integers may be written using bases from 2 to 36. For bases greater than ten, the characters 'a' through 'z' (case insensitive) represent digit values 10 through 35. The default base is decimal. A non-decimal number is prefixed by its base value, specified as a decimal number followed by either 'r' or 'R'.

Real numbers may be written in decimal only. The exponent of a floating-point format number indicates multiplication of the mantissa by 10 raised to the exponent power; i.e.,

$$\text{nnnnEddd} = \text{nnnn} \times 10^{\text{ddd}}$$

A number with a digit that is not appropriate for the base will cause a lexical error, as will an integer constant that is too large to be represented. If the absolute value of a real constant is too large or too small to be represented, the value of the constant will be \pm infinity or zero, respectively.

¹⁰ Unlike Smalltalk, integer literals are limited in range to smallInts.

¹¹ In situations where parsing the minus sign as part of the number would cause a parse error (for example, in the expression a-1), the minus is interpreted as a binary message (a - 1).

3.4.8 Comments

Comments are delimited by double quotes (""). Double quotes may not themselves be embedded in the body of a comment. All characters (including formatting characters like newline and carriage return) are part of the body of a comment.

Productions:

comment	→	"" { comment-char } ""
comment-char	→	any character except ""

Example:

```
"this is a comment"
```


THE SELF WORLD

The default Self world is a set of useful objects, including objects that can be used in application programs (e.g., integers, strings, and collections), objects that support the programming environment (e.g., the debugger), and objects that simply are used to organize the other objects. This document describes how this world is organized, focusing primarily on those objects meant for use in Self programs. It does not discuss the objects used to implement system facilities—for example, there is no discussion of the objects used to implement the graphical user interface—nor does it discuss how to use programming support objects such as the command history object; such tools are described in The Self User’s Manual.

The reader is assumed to be acquainted with the Self language, the use of multiple inheritance, the use of traits objects and prototype objects, and the organizing principles of the Self world as discussed in [UCC91].

4.1 World Organization

4.1.1 The Lobby

The lobby object is thus named because it is where objects enter the Self world. For example, when a script that creates a new object is read into the system, all expressions in that script are evaluated in the context of the lobby. That is, the lobby is the receiver of all messages sent to “self” by expressions in the script. To refer to some existing object in a script, the object must be accessible by sending a message to the lobby. For example, the expression:

```
_AddSlots: ( | newObject = ( | entries <- list copy ... | ) | )
```

requires that the message `list` be understood by the lobby (the implicit receiver of the message) so that the `entries` slot of the new object can be initialized. The lobby slots `traits`, `globals`, and `mixins` are the roots of the object namespaces accessible from the lobby. The organization of these namespaces is described in the next section. The slot `lobby` allows the lobby itself to be referred by name.

The lobby also has a number of other functions: it is the location of the default behavior inherited by most objects in the system (slot `defaultBehavior`).

4.1.2 Names and Paths

For convenience, the lobby’s namespace is broken into three pieces, implemented as separate objects rooted at the lobby:

traits objects that encapsulate shared behavior. Typically, each prototype object has an associated `traits` object of the same name that describes the shared part of its behavior.

globals prototypical objects and one-of-a-kind objects (“oddballs”)

mixins small, parentless bundles of behavior designed to be “mixed into” some other object

Each of these namespace objects is categorized to aid navigation.

For example, to find the parent of the prototype list object, one could start with the `globals` slot of the lobby, then get the `list` slot of that object, and then the `parent` slot of the list. The sequence of slot names, `globals list parent` is called a *path* and constitutes the list parent's *full name*. Parent slots can be omitted from an object's full name, since the slots in a parent are visible in the child via inheritance. A path with parent slots omitted forms the *short name* for an object. For example, the short name for the list parent is simply `list parent`.

Non-parent slots are used when it is desirable to keep a part of the name space distinct. For example, the `traits` slot of the lobby is not a parent slot. This allows a convention that gives prototypes and their associated traits objects similar names: a prototype and its associated traits object have the same local name, but the prototype is placed in a slot in the `globals` object, whereas the traits of the prototype is placed in a slot in the `traits` object. Since the `traits` slot of the lobby is not a parent slot, the name of the traits object must start with the prefix `traits`. The `globals` slot, on the other hand, is a parent slot, so the name of a prototype object needs no prefix. Thus, `list` refers to the prototype list while `traits list` refers to its traits object for lists.

As a matter of style, programs should refer to objects by the shortest possible name. This makes it easier to re-organize the global namespace as the system evolves. (If programs used full path names, then many more names would have to be updated to reflect changes to the namespace organization, a tedious chore.)

4.2 The Roots of Behavior

4.2.1 Default Behavior

Certain common behavior is shared by nearly all objects in the Self world. This basic behavior is defined in the `defaultBehavior` slot of the lobby and includes:

- identity comparisons (`==` and `!=='`)
- inequality (`!=='`)
- default behavior for printing (reimplement `printString` in descendants)
- mirror creation (`reflect:`)
- support for point, and list construction (`@` and `&`)
- behavior that allows blocks to ignore extra arguments
- behavior that allows an object to behave like a block that evaluates to that object (this permits a non-block object to be passed to a method that expects a block)
- behavior that allows an object to be its own key in a collection (`key`)
- default behavior for doubly-dispatched messages
- behavior for printing error messages and stack dumps (`error:` and `halt`)

It is important to note that not all objects in the system inherit this default behavior. It is entirely permissible to construct objects that do not inherit from the lobby, and the Self world contains quite a few such objects. For example, the objects used to break a namespace into separate categories typically do not inherit from the lobby. Any program intended to operate on arbitrary objects, such as a debugger, must therefore assume that the objects it manipulates do not understand even the messages in `defaultBehavior`.

Modules: `defaultBehavior`, `errorHandling`

4.2.2 The Root Traits: Traits Clonable and Traits Oddball

Most concrete objects in the Self world are descendants of one of two top-level traits objects: `traits clonable` and `traits oddball`. The distinction between the two is based on whether or not the object is *unique*. For example, `true` is a unique object. There is only one `true` object in the entire system, although there are many references to it. On the other hand, a list object is not unique. There may be many lists in the system, each containing different elements. A unique object responds to the message `copy` by returning itself and uses `identity` to test for equality. The general rule is:

- unique objects usually inherit from `traits oddball`
- non-unique objects usually inherit from `traits clonable`

Module: `rootTraits`

4.2.3 Mixins

Like traits objects, mixin objects encapsulate a bundle of shared behavior. Unlike traits objects, however, mixin objects are generally parentless to allow their behavior to be added to an object without necessarily also adding unwanted behavior (such as access to the lobby namespace). Mixins are generally used in objects that also have other parents. An example is `mixins identity`.

4.2.4 The Identity Mixin

Two objects are usually tested for equality based on whether they have “the same value” within a common domain. For example, $3.0 = 3$ within the domain of numbers, even though they are not the same object or even the same kind of object. In some domains, however, two objects are equal if and only if they are the exact same object. For example, even two process objects with the same state are not considered equal unless they are identical. In such cases, identity comparison is used to implement equality tests, and `mixins identity` can be mixed in to get the desired behavior.

Module: `rootTraits`

4.3 Blocks, Booleans, and Control Structures

A *block* is a special kind of object containing a sequence of statements. When a block is evaluated by being sent an acceptable `value` message, its statements are executed in the context of the current activation of the method in which the block is declared. This allows the statements in the block to access variables local to the block’s enclosing method and any enclosing blocks in that method. (This set of variables comprises the lexical scope of the block.) It also means that within the block, `self` refers to the receiver of the message that activated the method, not to the block object itself. A `return` statement in a block causes a return from the block’s enclosing method. (See chapter [pp-langref - Language Reference](#) for a more thorough discussion of block semantics.)

A block can take an arbitrary number of arguments and can have its own local variables, as well as having access to the local variables of its enclosing method. The statements in the block are executed when the block is sent a message of the form “`value[:{With:}]`”, where the number of colons in the message is at least the same as the number of arguments the block takes (extra arguments are ignored, but it is an error to provide too few). For example, the following block takes two arguments:

```
[| :arg1. :arg2 | arg1 + arg2 ]
```

and can be evaluated by sending it the message `value:With:` to produce the sum of its arguments. Blocks are used to implement all control structures in Self and allow the programmer to easily extend the system with customized

control structures. In fact, all control structures in Self except message sends, returns, and VM error handling are implemented using blocks.

4.3.1 Booleans and Conditionals

The fundamental control structure is the conditional. In Self, the behavior of conditionals is defined by two unique boolean objects, `true` and `false`. Boolean objects respond to the messages `ifTrue:`, `ifFalse:`, `ifTrue:False:`, and `ifFalse:True:` by evaluating the appropriate argument block. For example, `true` implements `ifTrue:False: as:`

```
ifTrue: b1 False: b2 = ( b1 value )
```

That is, when `true` is sent `ifTrue:False:`, it evaluates the first block and ignores the second. For example, the following expression evaluates to the absolute value of `x`:

```
x < 0 ifTrue: [ x negate ] False: [ x ]
```

The booleans also define behavior for the logical operations AND (`&&`), OR (`||`), EXCLUSIVE-OR (`^^`), and NOT (`not`). Because the binary boolean operators all send `value` to their argument when necessary, they can also be used for “short-circuit” evaluation by supplying a block, e.g.:

```
(0 <= i) && [ i < maxByte pred ] ifTrue: [...]
```

Module: `boolean`

4.3.2 Loops

The various idioms for constructing loops in Self are best illustrated by example. Here is an endless loop:

```
[ ... ] loop
```

Here are two loops that test for their termination condition at the beginning of the loop:

```
[ proceed ] whileTrue: [ ... ]  
[ quit ] whileFalse: [ ... ]
```

In each case, the block that receives the message repeatedly evaluates itself and, if the termination condition is not yet met, evaluates the argument block. The value returned by both loop expressions is `nil`.

It is also possible to put the termination test at the end of the loop, ensuring that the loop body is executed at least once:

```
[ ... ] untilTrue: [ quit ]  
[ ... ] untilFalse: [ proceed ]
```

Here is a loop that exits from the middle when `quit` becomes true:

```
[| :exit | ... quit ifTrue: exit ... ] loopExit
```

For the incurably curious: the parameter to the user’s block, supplied by the `loopExit` method, is simply a block that does a return from the `loopExit` method. Thus, the loop terminates when `exit value` is evaluated. The constructs `loopExitValue`, `exit`, and `exitValue` are implemented in a similar manner.

The value returned by the overall `[...] loopExit` expression is `nil`. Here is a loop expression that exits and evaluates to a value determined by the programmer when `quit` becomes true:

```
[| :exit | ... quit ifTrue: [ exit value: expr ] ] loopExitValue
```

Module: block

4.3.3 Block Exits

It is sometimes convenient to exit a block early, without executing its remaining statements. The following constructs support this behavior:

```
[| :exit | ... quit ifTrue: exit ... ] exit
[| :exit | ... quit ifTrue: [ exit value: expr ] ... ] exitValue
```

The first expression evaluates to nil if the block exits early; the second allows the programmer to define the expression's value when the block exits early. Note: These constructs should not be confused with their looping counterparts `loopExit` and `loopExitValue`.

Module: block

4.3.4 Other Block Behavior

Blocks have some other useful behavior:

- One can determine the time in milliseconds required to execute a block using various ways of measuring time using the messages `userTime`, `systemTime`, `cpuTime`, and `realTime`.
- One can profile the execution of a block using the messages `profile` and `flatProfile`. `profile` prints out the source level call graph annotated with call site and timing information whereas `flatProfile` prints out a flat profile sorted by module.
- The message `countSends` will collect lookup statistics during a block execution.

Any object that inherits from the lobby can be passed to a method that expects a block; behavior in `defaultBehavior` makes the object behave like a block that evaluates to that object.

Module: block

4.4 Numbers and Time

The SELF number traits form the hierarchy shown below. (In this and subsequent hierarchy descriptions, indentation indicates that one traits object is a child of another. The prefix “traits” is omitted since these hierarchy descriptions always describe the interrelationship between traits objects. In most cases, leaf traits are concrete and have an associated prototype with the same name.)

```
orderedOddball
  number
    float
    integer
      smallInt
      bigInt
```

`traits number` defines behavior common to all numbers, such as `successor`, `succ`, `predecessor`, `pred`, `absoluteValue`, `negate`, `double`, `half`, `max:`, and `min:`. `traits number` inherits from `traits orderedOddball`, so sending `copy` or `clone` to a number returns the number itself. `traits integer` defines behavior common to all integers such as `even`, `odd`, and `factorial`. There are four division operators for

integers that allow the programmer to control how the result is truncated or rounded. Integers also include behavior for iterating through a subrange, including:

```
to:Do:
to:By:Do:
to:ByNegative:Do:
upTo:Do:
upTo:By:Do:
downTo:Do:
downTo:By:Do:
```

Relevant oddballs:

- `infinity` IEEE floating-point infinity
- `minSmallInt` smallest `smallInt` in this implementation
- `maxSmallInt` biggest `smallInt` in this implementation

Modules: `number`, `float`, `integer`, `smallInt`, `bigInt`

4.4.1 Random Numbers

```
clonable
  random
    randomLC
      prototypes random
```

`traits random` defines the abstract behavior of random number generators. A random number generator can be used to generate random booleans, integers, floats, characters or strings. `traits randomLC` defines a concrete specialization based on a simple linear congruence algorithm. For convenience, the prototype for `randomLC` is “`random`,” not “`randomLC`”.

Modules: `random`

4.4.2 Time

```
clonable
  time
```

A time object represents a date and time (to the nearest millisecond) since midnight GMT on January 1, 1970. The message `current` returns a new time object containing the current time. Two times can be compared using the standard comparison operators. One time can be subtracted from another to produce a value in milliseconds. An offset in milliseconds can be added or subtracted from a time object to produce a new time object. However, it is an error to add two time objects together.

Modules: `time`

4.5 Collections

```
clonable
  collection
    ... collection hierarchy ...
```

Collections are containers that hold zero or more other objects. In Self, collections behave as if they have a key associated with each value in the collection. Collections without an obvious key, such as lists, use each element as both key and value. Iterations over collections always pass both the value and the key of each element (in that order) to the iteration block. Since Self blocks ignore extra arguments, this allows applications that don't care about keys to simply provide a block that takes only one argument.

Collections have a rich protocol. Additions are made with `at:Put:`, or with `add:` or `addAll:` for implicitly keyed collections. Iteration can be done with `do:` or with variations that allow the programmer to specify special handling of the first and/or last element. `with:Do:` allows pairwise iteration through two collections. The `includes:`, `occurrencesOf:`, and `findFirst:IfPresent:IfAbsent:` messages test for the presence of particular values in the collection. `filterBy:Into:` creates a new collection including only those elements that satisfy a predicate block, while `mapBy:Into:` creates a new collection whose elements are the result of applying the argument block to each element of the original collection.

Abstract collection behavior is defined in `traits collection`. Only a small handful of operations need be implemented to create a new type of collection; the rest can be inherited from `traits collection`. (See the `descendantResponsibility` slot of `traits collection`.) The following sections discuss various kinds of collection in more detail.

Modules: `collection` (abstract collection behavior)

4.5.1 Indexable Collections

```
collection
  indexable
    mutableIndexable
      byteVector
        ...the string hierarchy
      sequence
        sortedSequence
      vector
```

Indexable collections allow random access to their elements via keys that are integers. All sequences and vectors are indexable. The message `at:` is used to retrieve an element of an indexable collection while `at:Put:` is used to update an element of a `mutableIndexable` collection (other than a `sortedSequence`).

Modules: `indexable`, `abstractString`, `vector`, `sequence`, `sortedSequence`

4.5.2 Strings, Characters, and Paragraphs

```
collection
  ...
  byteVector
    string
      mutableString
      immutableString
      canonicalString
```

A string is a vector whose elements are character objects. There are three kinds of concrete string: immutable strings, mutable strings and canonical strings. `traits string` defines the behavior shared by all strings. A character is a string of length one that references itself in its sole indexable slot.

Mutable strings can be changed using the message `at:Put:`, which takes a character argument, or `at:PutByte:`, which takes an integer argument. An immutable string cannot be modified, but sending it the `copyMutable` message returns a mutable string containing the same characters.

Canonical strings are registered in a table inside the virtual machine, like Symbol objects in Smalltalk or atoms in LISP. The VM guarantees that there is at most one canonical string for any given sequence of bytes, so two canonical strings are equal (have the same contents) if and only if they are identical (are the same object). This allows efficient equality checks between canonical strings. All message selectors and string literals are canonical strings, and some primitives require canonical strings as arguments. Sending `canonicalize` to any string returns the corresponding canonical string.

Character objects behave like immutable strings of length one. There are 256 well-known character objects in the Self universe. They are stored in a 256-element vector named `ascii`, with each character stored at the location corresponding to its ASCII value. Characters respond to the message `asByte` by returning their ASCII value (that is, their index in `ascii`). The inverse of `asByte`, `asCharacter`, can be sent to an integer between 0 and 255 to obtain the corresponding character object.

Module: `string`

4.5.3 Unordered Sets and Dictionaries

```
collection
  setOrDictionary
    set
      sharedSet
  dictionary
    sharedDictionary
```

There are two implementations of sets and dictionaries in the system. The one described in this section is based on hash tables. The one discussed in the following section is based on sorted binary trees. The hash table implementation has better performance over a wide range of conditions. (An unfortunate ordering of element additions can cause the unbalanced trees used in the tree version to degenerate into an ordered lists, resulting in linear access times.)

A set behaves like a mathematical set. It contains elements without duplication in no particular order. A dictionary implements a mapping from keys to values, where both keys and values are arbitrary objects. Dictionaries implement the usual collection behavior plus keyed access using `at:` and `at:Put:` and the dictionary-specific operations `includesKey:` and `removeKey:.` In order to store an object in a set or use it as a dictionary key, the object must understand the messages `hash` and `=`, the latter applying to any pair of items in the collection. This is because sets and dictionaries are implemented as hash tables.

Derived from `set` and `dictionary` are `sharedSet` and `sharedDictionary`. These provide locking to maintain internal consistency in the presence of concurrency.

Modules: `setAndDictionary`, `sharedSetAndDictionary`

4.5.4 Tree-Based Sets and Dictionaries

```
collection
  tree
    treeNode abstract
      treeNode bag
      treeNode set
    emptyTrees abstract
      emptyTrees bag
      emptyTrees set
```

`treeSet` and `treeBag` implement sorted collections using binary trees. The set variant ignores duplicates, while the bag variant does not. Tree sets and bags allow both explicit and implicit keys (that is, adding elements can be done with either `at:Put:` or `add:`), where a tree set that uses explicit keys behaves like a dictionary. Sorting is done on

explicit keys if present, values otherwise, and the objects sorted must be mutually comparable. Comparisons between keys are made using `compare:IfLess:Equal:Greater:`.

The implementation of trees uses dynamic inheritance to distinguish the differing behavior of empty and non-empty subtrees. The prototype `treeSet` represents an empty (sub)tree; when an element is added to it, its parent is switched from `traits emptyTrees set`, which holds behavior for empty (sub)trees, to a new copy of `treeSetNode`, which represents a tree node holding an element. Thus, the `treeSet` object now behaves as a `treeSetNode` object, with right and left subtrees (initially copies of the empty subtree `treeSet`). Dynamic inheritance allows one object to behave modally without using clumsy if-tests throughout every method.

One caveat: since these trees are not balanced, they can degenerate into lists if their elements are added in sorted order. However, a more complex tree data structure might obscure the main point of this implementation: to provide a canonical example of the use of dynamic inheritance.

Modules: `tree`

4.5.5 Lists and PriorityQueues

```
collection
  list
  priorityQueue
```

A list is an unkeyed, circular, doubly-linked list of objects. Additions and removals at either end are efficient, but removing an object in the middle is less so, as a linear search is involved.

A `priorityQueue` is an unkeyed, unordered collection with the property that the element with the highest priority is always at the front of the queue. Priority queues are useful for sorting (heapsort) and scheduling. The default comparison uses `<`, but this can be changed.

Modules: `list`, `priorityQueue`

4.5.6 Constructing and Concatenating Collections

```
clonable
  collector
```

Two kinds of objects play supporting roles for collections. A `collector` object is created using the `&` operator (inherited from `defaultBehavior`), and represents a collection under construction. The `&` operator provides a concise syntax for constructing small collections. For example:

```
(1 & 'abc' & x) asList
```

constructs a list containing an integer, a string, and the object `x`. A `collector` object is not itself a collection; it is converted into one using a conversion message such as `asList`, `asVector`, or `asString`.

Modules: `collector`

4.6 Pairs

```
pair
  point
  rectangle
```

`traits pair` describes the general behavior for pairs of arithmetic quantities. A point is a pair of numbers representing a location on the cartesian plane. A rectangle is a pair of points representing the opposing corners of a rectangle whose sides are parallel with the x and y axes.

Modules: `pair`, `point`, `rectangle`

4.7 Mirrors

```
collection
  mirror
    mirrors smallInt
    mirrors float
    mirrors vectorish
      mirrors vector
      mirrors byteVector
        mirrors canonicalString
        mirrors mirror
    mirrors block
    mirrors method
    mirrors blockMethod
      mirrors activation liveOnes
        mirrors activation
          mirrors deadActivation
          mirrors methodActivation
          mirrors blockMethodActivation
    mirrors process
    mirrors assignment
    mirrors slots
    mirrors profiler
```

Mirrors allow programs to examine and manipulate objects. (Mirrors get their name from the fact that a program can use a mirror to examine—that is, reflect upon—itsself.) A mirror on an object `x` is obtained by sending the message `reflect : x` to any object that inherits `defaultBehavior`. The object `x` is called the mirror’s *reflectee*. A mirror behaves like a keyed collection whose keys are slot names and whose values are mirrors on the contents of slots of the reflectee. A mirror can be queried to discover the number and names of the slots in its reflectee, and which slots are parent slots. A mirror can be used to add and remove slots of its reflectee. Iterating through a mirror enumerates objects representing slots of the reflected object (such facets are called “fake” slots). For example, a method mirror includes fake slots for the method’s byte code and literal vectors and elements of vectors and `byteVectors`.

There is one kind of mirror for each kind of object known to the virtual machine: small integers, floats, canonical strings, object and byte vectors, mirrors, blocks, ordinary and block methods, ordinary and block method activations, processes, profilers, the assignment primitive, and ordinary objects (called “slots” because an ordinary object is just a set of slots). The prototypes for these mirrors are part of the initial Self world that exists before reading in any script files. The file `init.self` moves these prototypes to the `mirrors` subcategory of the `prototypes` category of the `lobby` namespace. Because `mirrors` is not a parent slot, the names of the mirror prototypes always include the “mirrors” prefix.

Modules: `mirror`, `slot`, `init`

4.8 Messages

Self allows messages to be manipulated as objects when convenient. For example, if an object fails to understand a message, the object is notified of the problem via a message whose arguments include the selector of the message that was not understood. While most objects inherit default behavior for handling this situation (by halting with an error),

it is sometimes convenient for an object to handle the situation itself, perhaps by resending the message to some other object. Objects that do this are called transparent forwarders. An example is given in `interceptor`.

A string has the basic ability to use itself as a message selector using the messages `sendTo:` (normal message sends), `resendTo:` (resends), or `sendTo:DelegatingTo:` (delegated sends). Each of these messages has a number of variations based on the number of arguments the message has. For example, one would use `sendTo:With:With:` to send a message with `at:Put:` as the selector and two arguments:

```
'at:Put:' sendTo: aDict With: k With: v
```

Note: Primitives such as `_Print` cannot be sent in the current system.

A selector, receiver, delegatee, methodHolder, and arguments can be bundled together in a `message` object. The message gets sent when the message object receives the `send` message. Message objects are used to describe delayed actions, such as the actions that should occur just before or after a snapshot is read. They are also used as an argument to new process creation (you can create a new process to execute the message by sending it `fork`).

Modules: `sending`, `message`, `selector`, `interceptor`

4.9 Processes and the Prompt

Self processes are managed by a simple preemptive round-robin scheduler. Processes can be stepped, suspended, resumed, terminated, or put to sleep for a specified amount of time. Also, the stack of a suspended process can be examined and the CPU use of a process can be determined. A process can be created by sending `fork` to a message.

The `prompt` object takes input from `stdin` and spawns a process to evaluate the message. Input to the prompt is kept in a history list so that past input can be replayed, similar to the history mechanism in many Unix shells.

Modules: `process`, `scheduler`, `semaphore`, `prompt`, `history`

4.10 Foreign Objects

```
clonable
  proxy
    fctProxy
      foreignFct
    foreignCode
```

The low level aspects of interfacing with code written in other languages (via C or C++ glue code) are described in *Virtual Machine Reference (chapter 8)*. A number of objects in the Self world are used to interface to foreign data objects and functions. These objects are found in the name spaces `traits foreign`, and `globals foreign`.

One difficulty in interfacing between Self and external data and functions is that references to foreign data and functions from within Self can become obsolete when the Self world is saved as a snapshot and then read in later, possibly on some other workstation. Using an obsolete reference (i.e., memory address) would be disastrous. Thus, Self encapsulates such references within the special objects `proxy` (for data references) and `fctProxy` (for function references). Such objects are known collectively as *proxies*. A proxy object bundles some extra information along with the memory address of the referenced object and uses this extra information to detect (with high probability) any attempt to use an obsolete proxy. An obsolete proxy is called a *dead proxy*.

To make it possible to rapidly develop foreign code, the virtual machine supports dynamic linking of this code. This makes it unnecessary to rebuild the virtual machine each time a small change is made to the foreign code. Dynamic

linking facilities vary from platform to platform, but the Self interface to the linking facilities is largely system independent. The SunOS/Solaris dynamic link interface is defined in the `sunLinker` object. However, clients should always refer to the dynamic linking facilities by the name `linker`, which will be initialized to point to the dynamic linker interface appropriate for the current platform.

The `linker`, `proxy` and `fctProxy` objects are rather low level and have only limited functionality. For example, a `fctProxy` does not know which code file it is dependent on. The objects `foreignFct` and `foreignCode` establish a higher level and easier to use interface. A `foreignCode` object represents an “object file” (a file with executable code). It defines methods for loading and unloading the object file it represents. A `foreignFct` object represents a foreign routine. It understands messages for calling the foreign routine and has associated with it a `foreignCode` object. The `foreignFct` and `foreignCode` objects cooperate with the linker, to ensure that object files are transparently loaded when necessary and that `fctProxies` depending on an object file are killed when the object file is unloaded, etc.

The `foreignCodeDB` object ensures that `foreignCode` objects are unique, given a path. It also allows for specifying initializers and finalizers on `foreignCode` objects. An initializer is a foreign routine that is called whenever the object file is loaded. Initializers take no arguments and do not return values. Typically, they initialize global data structures. Finalizers are called when an object file is unloaded. When debugging foreign routines, `foreignCodeDB` `printStatus` outputs a useful overview.

Normal use of a foreign routine simply involves cloning a `foreignFct` object to represent the foreign routine. When cloning it, the name of the function and the path of the object file is specified. It is then not necessary to worry about `proxy`, `fctProxy` and `linker` objects, etc. In fact, it is recommended *not* to send messages directly to these objects, since this may break the higher level invariants that `foreignFct` objects rely on.

Relevant oddballs:

<code>linker</code>	Dynamic linker for current platform.
<code>sunLinker</code>	Dynamic linker implementation for SunOS/Solaris.
<code>foreignCodeDB</code>	Registry for <code>foreignCode</code> objects.

Modules: `foreign`

4.11 I/O and Unix

```

oddball
  unix
clonable
  proxy
  unixFile (mixes in traits unixFile currentOsVariant)
    
```

Warning: This page is out of date for Self 4.5.

Start looking at the object `os` instead of `unix`.

Note: If reading from `stdin`, the `prompt` object may interfere with your code by stealing input from you. To avoid this, wrap calls in `prompt suspendWhile: []`, for example:

```
prompt suspendWhile: [ stdin readLine printLine ]
```

which will read a line from the `stdin` and echo it to `stdout`.

The oddball object `unix` provides access to selected Unix system calls. The most common calls are the file operations: `creat()`, `open()`, `close()`, `read()`, `write()`, `lseek()` and `unlink()`. `tcpConnectToHost:Port:IfFail:` opens a TCP connection. The `select()` call and the indirect system call are also supported (taking a variable number of integer, float or byte vector arguments, the latter being passed as C pointers). `unixFile` provides a higher level interface to the Unix file operations. The oddball object `tty` implements terminal control facilities such as cursor positioning and highlighting.

Relevant oddballs:

<code>stdin</code> , <code>stdout</code> , <code>stderr</code>	standard Unix streams
<code>tty</code>	console terminal capabilities

Modules: `unix`, `stdin`, `tty`, `ttySupport`, `termcap`

4.12 Other Objects

Here are some interesting oddball objects not discussed elsewhere:

<code>comparator</code>	An object that can compute “diffs” between sequences.
<code>compilerProfiling</code>	Compiler profiling.
<code>desktop</code>	The controlling object for the graphical user interface.
<code>history</code>	A history of commands typed at the prompt, and their results.
<code>memory</code>	Memory system interface (GC, snapshot, low space, etc.).
<code>monitor</code>	System monitor (spy) control.
<code>nil</code>	Indicates an uninitialized value.
<code>platforms</code>	Possible hardware platforms.
<code>preferences</code>	User configuration preferences.
<code>profiling</code> , <code>flatProfiling</code>	Controls Self code profiling.
<code>prompt</code>	Interactive read-eval-print loop.
<code>scheduler</code>	Self process scheduler.
<code>snapshotAction</code>	Actions to do before/after a snapshot.
<code>thisHost</code>	Describes the current host platform.
<code>times</code>	Reports user, system, cpu, or real time.
<code>typeSizes</code>	Bit/byte sizes for primitive types.
<code>vmProfiling</code>	Virtual machine profiling.

4.13 How to build the world

Last updated 2 February 2016 for Self 4.6.0

Should you need to reconstruct a world from the source files, here’s how to do it. This section describes how to create a default object world by reading in the Self source code distributed through the [GitHub repository](#). You can also do this after writing the world out using the transporter (`transporter fileOut fileOutAll`).

4.13.1 From within the main Self tree

To create the default object world, change your current working directory to the `objects` subdirectory of the Self source release and follow these steps:

1. Start the Self VM:

```
% Self
Self Virtual Machine Version 4.1.13, Sat 04 Jan 14 12:17:37 Mac OS X i386 (4.4-
↪268-g58a0717)
Copyright 1989-2003: The Self Group (type _Credits for credits)

VM#
```

2. (Optional, but recommended.) Start the spy so you can watch the world fill up with objects:

```
VM# _Spy: true
```

Note: You must use the primitive to do this because the world is empty.

3. Read in the default world. To do this, ask Self to read expressions from a file:

```
VM# 'worldBuilder.self' _RunScript
```

4.13.2 From outside the main tree

When developing applications which aren't part of the main Self distribution, it is often convenient to build a Self world from a directory other than the default directory. You can specify on the command line where the main Self distribution is and the options. For example the below builds the world with morphic and also ui1. Use `-o none` to build without any graphics.

```
% Self -f /path/to/Self/objects/worldBuilder.self \
      -b /path/to/Self/objects \
      -o morphic,ui1
```

4.13.3 Finishing the build

Once you have started the `worldBuilder.self` script, you will be given options as to which features you would like in your new world.

1. Unless you have asked Self not to print script names, you should see something like:

```
Reading worldBuilder.self...
reading ./core/init.self...
reading ./core/allCore.self...
reading ./core/systemStructure.self...
. . .
```

2. Unless you have specified the options on the command line, then at various places, you will be asked if you wish to add optional additions to the base system, such as the morphic user interface (UI2) or the earlier UI1 (which requires X11 to run):

```
Load UI2 (Morphic)? (y/N)
> y
```

3. After all the files have been read in, Self will start the process scheduler, initialize its module cache, and print:

```
"Self 0"
```

That last line is the Self prompt indicating that the system is ready to read and evaluate expressions.

4. If you have loaded Morpich, you may wish to open up a window:

```
"Self 0" desktop open
Adjusting VM for better UI2 performance:
_MaxPICSize: 25
_Flush

The ui2 desktop is now running. Type:
"desktop stop" to suspend it,
"desktop go" to resume it after stopping, and
"desktop close" to close it.

desktop
"Self 1"
```

4.14 How to use the low-level interrupt facilities

There are two low-level ways to interrupt a running Self program¹, Control-C and Control-\. The second way works even if the Self process scheduler is not running. In response to the interrupt, you will see one of two things. If the Self scheduler is not running, you will be returned directly to the VM# prompt. If the scheduler is running, you will be presented with a list of Self processes (the process menu):

```
Self 9> 100000 * 100000 do: []
^C
-----Interrupt-----
Ready:
  <25> scheduling process 100000 * 100000 do: []
-----
Select a process (or q to quit scheduler): 25
Select <return> for no action
  p to print the stack
  k to kill the process
  b to resume execution of the process in the background
  s to suspend execution of the process
for process 25: k
Process 25 killed.
-----
Self 10>
```

In this example, the loop was interrupted by typing Control-C, and the process menu was used to abort the process. If the user had typed “q” to quit the scheduler, all current processes would have been aborted along with the scheduler itself:

```
...
-----
Select a process (or q to quit scheduler): q
Scheduler shut down.
-----
prompt
VM#
```

The scheduler has been stopped, returning the user to the VM# prompt. The command prompt start restarts the scheduler:

¹ Normally, you would use debugging facilities provided in the programming environment.

```
VM# prompt start
Self 11>
```

Although the VM# prompt can be used to evaluate expressions directly, the scheduler supports much nicer error messages and debugging, so it is usually best to run the scheduler. (The scheduler is started automatically when the default world is created.)

Certain virtual machine operations like garbage collection, reading a snapshot, and compilation cannot be interrupted; interrupts during these operations will be deferred until the operation is complete. As a last resort (e.g., if the system appears to be “hung”), you can force an abort by pressing Control-\ five times in a row.

4.15 Using the textual debugger

If you are modifying the core of the programming environment or working without the environment you may need to use the textual debugger. After attaching the aborted process to the debugger using the shell command `attach`, these commands are available:

Command	Description
<code>attach: n</code>	attach the process with object reference number <code>n</code>
<code>detach</code>	detach the debugged process
<code>step[:n]</code>	execute (n) non trivial bytecodes *
<code>stepi[:n]</code>	execute (n) bytecodes
<code>next[:n]</code>	execute (n) non trivial bytecodes in the current activation
<code>nexti[:n]</code>	execute (n) bytecodes in the current activation
<code>finish</code>	finish executing the current activation
<code>cont</code>	continue execution
<code>trace</code>	print out a stack trace of the process
<code>show</code>	display the current activation
<code>show: n</code>	go to and display the nth activation on the stack
<code>status</code>	display the status of the debugged process
<code>up[: n]</code>	go up (n) activation(s)
<code>upLex</code>	go up to the lexical enclosing scope of this activation
<code>down[: n]</code>	go down (n) activation(s)
<code>lookup: <name></code>	lookup the given name in the context of the current activation

* A bytecode is trivial if it is a push of a literal or a send to a slot residing in the lexical scope of the current activation.

4.16 Logging

`log` is a useful system-wide logging mechanism. You can find it in the `system` category of `globals`.

4.16.1 How to log

There are a number of useful messages in the `logging` category of `log` which allow you to simply and cleanly log messages. For example:

```
log warn: 'This is a warning.'
```

You can log with one of five levels found at `log levels`. These are, in order of severity, `debug`, `info`, `warn`, `error`, `fatal`.

You can also tag log entries, for example:

```
log fatal: 'The server has caught fire' For: 'webserver'
```

By default, entries of either error or fatal severity which aren't tagged are logged to stderr in the form:

```
[Thu Oct 23 16:25:07 2014] error -- Something went wrong!
```

4.16.2 How logging works

The helper methods shown above construct a `log entry` and hand it to the `log dispatcher`. The dispatcher has a number of handlers, each is given a chance to handle the log entry. The handlers can choose which entries to act on. Example handlers are in `log prototypeHandlers`.

When making a handler, please keep in mind that the log entry's `message` is expected to be something which understands `value`, returning an object (or itself) which understands `asString`. If you do not need to resolve the message by sending it `value` please don't; that way logs can be sent blocks which are only resolved if necessary; eg:

```
log debug: ['We have reached: ', somethingComplicatedToCalculate]
```

will not slow down your code if no log handler is interested in handling debuggers.

If your handler breaks the logging process you can restart it by:

```
log dispatcher hup
```


A GUIDE TO PROGRAMMING STYLE

This section discusses some programming idioms and stylistic conventions that have evolved in the Self group. Rather than simply presenting a set of rules, an attempt has been made to explain the reasons for each stylistic convention. While these conventions have proven useful to the Self group, they should be taken as guidelines, not commandments. Self is still a young language, and it is likely that its users will continue to discover new and better ways to use it effectively.

5.1 Behaviorism versus Reflection

One of the central principles of Self is that an object is completely defined by its behavior: that is, how it responds to messages. This idea, which is sometimes called *behaviorism*, allows one object to be substituted for another without ill effect—provided, of course, that the new object’s behavior is similar enough to the old object’s behavior. For example, a program that plots points in a plane should not care whether the points being plotted are represented internally in cartesian or polar coordinates as long as their external behavior is the same. Another example arises in program animation. One way to animate a sorting algorithm is to replace the collection being sorted with an object that behaves like the original collection but, as a side effect, updates a picture of itself on the screen each time two elements are swapped. Behaviorism makes it easier to extend and reuse programs, perhaps even in ways that were not anticipated by the program’s author.

It is possible, however, to write non-behavioral programs in Self. For example, a program that examines and manipulates the slots of an object *directly*, rather than via messages, is not behavioral since it is sensitive to the internal representation of the object. Such programs are called *reflective*, because they are reflecting on the objects and using them as data, rather than using the objects to represent something else in the world. Reflection is used to talk about an object rather than talking to it. In Self, this is done with objects called *mirrors*. There are times when reflection is unavoidable. For example, the Self programming environment is reflective, since its purpose is to let the programmer examine the structure of objects, an inherently reflective activity. Whenever possible, however, reflective techniques should be avoided as a matter of style, since a reflective program may fail if the internal structure of its objects changes. This places constraints on the situations in which the reflective program can be reused, limiting opportunities for reuse and making program evolution more difficult. Furthermore, reflective programs are not as amenable to automatic analysis tools such as application extractors or type inferencers.

Programs that depend on object *identity* are also reflective, although this may not be entirely obvious. For example, a program that tests to see if an object is identical to the object `true` may not behave as expected if the system is later extended to include fuzzy logic objects. Thus, like reflection, it is best to avoid using object identity. One exception to this guideline is worth mentioning. When testing to see if two collections are equal, observing that the collections are actually the same object can save a tedious element-by-element comparison. This trick is used in several places in the Self world. Note, however, that object identity is used only as a hint; the correct result will still be computed, albeit more slowly, if the collections are equal but not identical.

Sometimes the implementation of a program requires reflection. Suppose one wanted to write a program to count the number of unique objects in an arbitrary collection. The collection could, in general, contain objects of different, possibly incomparable, types. In Smalltalk, one would use an `IdentitySet` to ensure that each object was counted

exactly once. IdentitySets are reflective, since they use identity comparisons. In Self, the preferred way to solve this problem is to make the reflection explicit by using mirrors. Rather than adding objects to an IdentitySet, mirrors on the objects would be added to an ordinary set. This substitution works because two mirrors are equal if and only if their reflectees are identical.

In short, to maximize the opportunities for code reuse, the programmer should:

- avoid reflection when possible,
- avoid depending on object identity except as a hint, and
- use mirrors to make reflection explicit when it is necessary.

5.2 Objects Have Many Roles

Objects in Self have many roles. Primarily, of course, they are the elements of data and behavior in programs. But objects are also used to factor out shared behavior, to represent unique objects, to organize objects and behavior, and to implement elegant control structures. Each of these uses are described below.

5.2.1 Shared Behavior

Sometimes a set of objects should have the same behavior for a set of messages. The slots defining this *shared behavior* could be replicated in each object but this makes it difficult to ensure the objects continue to share the behavior as the program evolves, since the programmer must remember to apply the same changes to all the objects sharing the behavior. Factoring out the shared behavior into a separate object allows the programmer to change the behavior of the entire set of objects simply by changing the one object that implements the shared behavior. The objects that share the behavior inherit it via parent slots containing (references to) the shared behavior object.

By convention, two kinds of objects are used to hold shared behavior: *traits* and *mixins*. A traits object typically has a chain of ancestors rooted in the lobby. A mixin object typically has no parents, and is meant to be used as an additional parent for some object that already inherits from the lobby.

5.2.2 One-of-a-kind Objects (Oddballs)

Some objects, such as the object `true`, are unique; it is only necessary to have one of them in the system. (It may even be important that the system contain *exactly* one of some kind of object.) Objects playing the role of unique objects are called *oddballs*. Because there is no need to share the behavior of an oddball among many instances, there is no need for an oddball to have separate traits and prototype objects. Many oddballs inherit a `copy` method from `traits` `oddball` that returns the object itself rather than a new copy, and most oddballs inherit the global namespace and default behavior from the lobby.

5.2.3 Inline Objects

An inline object is an object that is nested in the code of a method object. The inline object is usually intended for localized use within a program. For example, in a finite state machine implementation, the state of the machine might be encoded in a selector that would be sent to an inline object to select the behavior for the next state transition:

```
state sendTo: (|
  inComment: c = ( c = '' ifTrue: [state: 'inCode']. self ).
  inCode: c = ( c = '' ifTrue: [state: 'inComment']
              False: ... )
|)
With: nextChar
```

In this case, the inline object is playing the role of a case statement.

Another use of inline objects is to return multiple values from a method, as discussed in section [5.4 How to Return Multiple Values](#). Yet another use of inline objects is to parameterize the behavior of some other object. For example, the predicate used to order objects in a *priorityQueue* can be specified using an inline object:

```
queue: priorityQueue copyRemoveAll.
queue sorter: (| element: e1 Precedes: e2 = ( e1 > e2 ) |).
```

(A block cannot be used here because the current implementation of Self does not support non- LIFO blocks, and the sorter object may outlive the method that creates it). There are undoubtedly other uses of inline objects. Inline objects do not generally inherit from the lobby.

5.3 Naming and Printing

When debugging or exploring in the Self world, one often wants to answer the question: “what is that object?” The Self environment provides two ways to answer that question. First, many objects respond to the `printString` message with a textual description of themselves. This string is called the object’s *printString*. An object’s `printString` can be quite detailed; standard protocol allows the desired amount of detail to be specified by the requestor. For example, the `printString` for a collection might include the `printStrings` of all elements or just the first few. Not all objects have `printStrings`, only those that satisfy the criteria discussed in section [5.3.2 How to make an object print](#) below.

The second way to describe an object is to give its *path name*. A path name is a sequence of unary selectors that describes a path from the lobby to the object. For example, the full path name of the prototype list is “globals list.” A path name is also an expression that can be evaluated (in the context of the lobby) to produce the object. Because “globals” is a parent slots, it can be omitted from this path name expression. Doing this yields the short path name “list.” Not all objects have path names, only those that can be reached from the lobby. Such objects are called *well-known*.

5.3.1 How objects are printed

When an expression is typed at the prompt, it is evaluated to produce a result object. The prompt then creates a mirror on this result object and asks the mirror to produce a name for the object. (A mirror is used because naming is reflective.) The object’s creator path annotation provides a hint about the path from the lobby to either the object itself or its prototype. If the object is a clone “a” or “an” is prepended to its prototype’s creator path. In addition to its path, the mirror also tries to compute a `printString` for the object if it is annotated as `isComplete`. Then, the two pieces of information are merged. For example, the name of the prototype list is “list” but the name of `list copy add: 17` is “a list(17).” See the naming category in mirror traits for the details of this process.

5.3.2 How to make an object print

The distinction between objects that hold shared behavior (traits and mixin objects) and concrete objects (prototypes, copies of prototypes, and oddballs) is purely a matter of convention; the Self language makes no such distinction. While this property (not having special kinds of objects) gives Self great flexibility and expressive power, it leads to an interesting problem: the inability to distinguish behavior that is ready for immediate use from that which is defined only for the benefit of descendant objects. Put another way: Self cannot distinguish those objects playing the role of classes from those playing the role of instances.

The most prominent manifestation of this problem crops up in object printing. Suppose one wishes to provide the following `printString` method for all point objects:

```
printString = ( x printString, '@', y printString )
```

Like other behavior that applies to all points, the method should be put in point traits. But what happens if `printString` is sent to the object `traits point`? The `printString` method is found but it fails when it attempts to send `x` and `y` to itself because these slots are only defined in point objects (not the `traits point` object). Of course there are many other messages defined in `traits point` that would also fail if they were sent to `traits point` rather than to a point object. The reason printing is a bigger problem is that it is useful to have a general object printing facility to be used during debugging and system exploration. To be as robust as possible, this printing facility should not send `printString` when it will fail. Unfortunately, it is difficult to tell when `printString` is likely to fail. Using reflection, the facility can avoid sending `printString` to objects that do not define `printString`. But that is not the case with `traits point`. The solution taken in this version of the system is to mark printable objects with a special annotation. The printing facility sends `printString` to the object only if the object contains an annotation `isComplete`.

The existence of an `isComplete` annotation in an object means that the object is prepared to print itself. The object agrees to provide behavior for a variety of messages; see the programming environment manual for more details.

5.4 How to Return Multiple Values

Sometimes it is natural to think of a method as returning several values, even though Self only allows a method to return a single object. There are two ways to simulate methods that return multiple values. The first way is to use an inlined object. For example, the object:

```
(| p* = lobby. lines. words. characters |)
```

could be used to package the results of a text processing method into a single result object:

```
count = (  
  | r = (| p* = lobby. lines. words. characters |) ... |  
  ...  
  r: r copy.  
  r lines: lCount. r words: wCount. r characters: cCount.  
  r )
```

Note: that the inline object prototype inherits `copy` from the `lobby`. If one omitted its parent slot `p`, one would have to send it the `_Clone` primitive to copy it. It is considered bad style, however, to send a primitive directly, rather than calling the primitive's wrapper method.

The sender can extract the various return values from the result object by name.

The second way is to pass in one block for each value to be returned. For example:

```
countLines:[| :n | lines: n ]  
  Words:[| :n | words: n ]  
  Characters:[| :n | characters: n ]
```

Each block simply stores its argument into the a local variable for later use. The `countLines:Words:Characters:` method would evaluate each block with the appropriate value to be returned:

```
countLines: lb Words: wb Characters: cb = (  
  ...  
  lb value: lineCount.  
  wb value: wordCount.  
  cb value: charCount.  
  ...
```

5.5 Substituting Values for Blocks

The lobby includes behavior for the block evaluation messages. Thus, any object that inherits from the lobby can be passed as a parameter to a method that expects a block — the object behaves like a block that evaluates that object. For example, one may write:

```
x >= 0 ifTrue: x False: x negate
```

rather than:

```
x >= 0 ifTrue: [ x ] False: [ x negate ]
```

Note: however, that Self evaluates all arguments before sending a message. Thus, in the first case `x negate` will be evaluated regardless of the value of `x`, even though that argument will not be used if `x` is nonnegative. In this case, it doesn't matter, but if `x negate` had side effects, or if it were very expensive, it would be better to use the second form.

In a similar vein, blocks inherit default behavior that allows one to provide a block taking fewer arguments than expected. For example, the collection iteration message `do:` expects a block taking two arguments: a collection element and the key at which that element is stored. If one is only interested in the elements, not the keys, one can provide a block taking only one argument and the second block argument will simply be ignored. That is, you can write:

```
myCollection do: [| :el | el printLine]
```

instead of:

```
myCollection do: [| :el. :key | el printLine]
```

5.6 nil Considered Naughty

As in Lisp, Self has an object called `nil`, which denotes an undefined value. The virtual machine initializes any uninitialized slots to this value. In Lisp, many programs test for `nil` to find the end of a list, or an empty slot in a hash table, or any other undefined value. There is a better way in Self. Instead of testing an object's identity against `nil`, define a new object with the appropriate behavior and simply send messages to this object; Self's dynamic binding will do the rest. For example, in a graphical user interface, the following object might be used instead of `nil`:

```
nullGlyph = (|
    display = ( self ).
    boundingBox = (0@0) # (0@0).
    mouseSensitive = false.
|)
```

To make it easier to avoid `nil`, the methods that create new vectors allow you to supply an alternative to `nil` as the initial value for the new vector's elements (e.g., `copySize:FillingWith:`).

5.7 Hash and =

Sets and dictionaries are implemented using hash tables. In order for an object to be eligible for inclusion in a set or used as a key in a dictionary, it must implement both `=` and `hash`. (`hash` maps an object to a `smallInt`.) Further,

hash must be implemented in such a way that for objects *a* and *b*, (*a* = *b*) implies (*a* hash = *b* hash). The behavior that sets disallow duplicates and dictionaries disallow multiple entries with the same key is dependent upon the correct implementation of hash for their elements and keys. Finally, the implementation of sets (and dictionaries) will only work if the hash value of the objects in the set do not change while the objects are in the set (dictionary). This may complicate managing sets of mutable objects, since if the hash value depends on the mutable state, the objects can not be allowed to mutate while in the set.

Of course, a trivial hash function would simply return a constant regardless of the contents of the object. However, for good hash table performance, the hash function should map different objects to different values, ideally distributing possible object values as uniformly as possible across the range of small integers.

5.8 Equality, Identity, and Indistinguishability

Equality, *identity*, and *indistinguishability* are three related concepts that are often confused. Two objects are *equal* if they “mean the same thing”. For example, `3 = 3.0` even though they are different objects and have different representations. Two objects are *identical* if and only if they are the same object. (Or, more precisely, two references are identical if they refer to the same object.) The primitive `_Eq`: tests if two objects are identical. Finally, two objects are *indistinguishable* if they have exactly the same behavior for every possible sequence of non-reflective messages. The binary operator “`==`” tests for indistinguishability. Identity implies indistinguishability which implies equality.

It is actually not possible to guarantee that two different objects are indistinguishable, since reflection could be used to modify one of the objects to behave differently after the indistinguishability test was made. Thus, `==` is defined to mean identity by default. Mirrors, however, override this default behavior; (`m1 == m2`) if (`m1 reflectee _Eq: m2 reflectee`). This makes it appear that there is at most one mirror object for each object in the system. This illusion would break down, however, if one added mutable state to mirror objects.

HOW TO PROGRAM IN SELF

6.1 Introduction

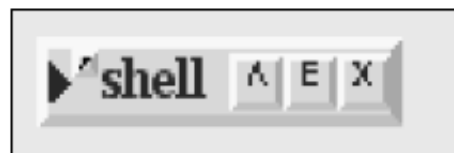
The Self programming environment provides facilities for writing programs, and the transporter provides a way to save them as source files. Of all the parts of Self, the programming environment probably has the least research ambition in it. We simply needed to concentrate the innovation in other areas: language design, compiler technology, user interface. The Self programming environment strives to meet the high standard set by Smalltalk's, but with a more concrete feels. The transporter, on the other hand, is somewhere in-between completely innovative research and dull development. It attempts to pull off a novel feat—programming live objects instead of text—and partially succeeds. Its novelty lies in its view of programs as collections of slots, not objects or classes, and its extraction of the programmer's intentions from a web of live objects.

On the Macintosh, Self uses option-click for a middle-mouse click, and uses command- (the apple key) click for the right button click. So wherever the text says “left-button-click” just click with the mouse, where it says “middle-button click” hold down the option key and click with the mouse, and where it says “right button click” hold down the command key and click with the mouse. These mappings are defined in Self, so you can change them by editing the `whichButton:` method in the `initialization` category in `traits ui2MacEvent`.

6.2 Browsing Concepts

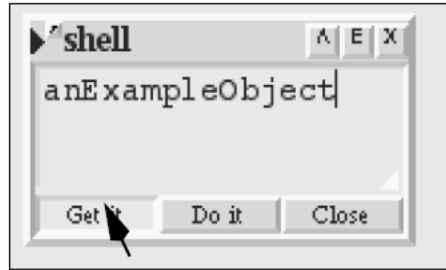
6.2.1 Introducing the Outliner

Objects in the Self environment are represented as *outliners*, which can expand to show increasing levels of detail. One of these objects has been designed to provide a convenient context for typed-in commands, and so it is called the shell. If the shell is not already present on your screen, you can summon it by pressing the middle mouse button on the background and selecting `shell`.



Outliners sport three small buttons in the top-right-hand corner labeled “^”, “E”, and “X”. These buttons summon the object's parents, add an evaluator text region to the bottom of the outliner, and dismiss the outliner. Press the “E” button to get an evaluator.

Type `anExampleObject` into the evaluator (it will already be selected) and hit the *Get it* button (or type `metareturn` on UNIX, or `command-return` on MacOS X):

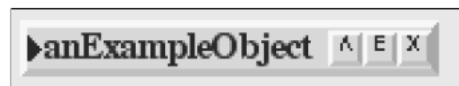


The result object appears in your “hand” raised above the screen as if you were dragging it with the left button.

Note: If this returns an error, it may be that you don’t have `anExampleObject` in your Self world. You can always load it from the file `objects/misc/programmingExamples.self` as follows:

```
'path/to/objects/misc/programmingExamples.self' runScript
```

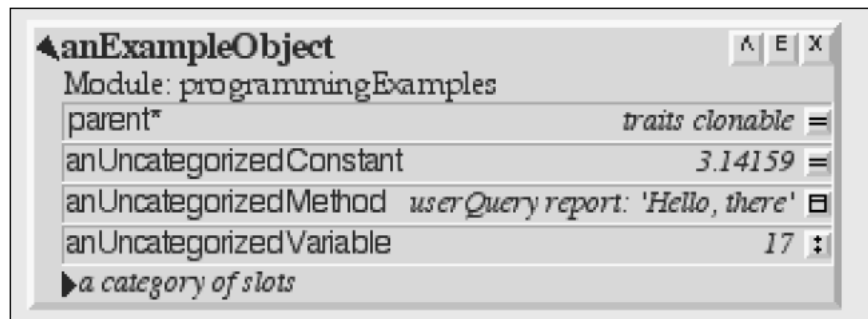
Click the button to set it down.



As with most other things on the Self screen, the left button picks it up and moves it. (For buttons and other things that use left-button for other purposes, you can grab them with marquee selection (really the *carpet morph* in Self) or with the “Grab” item on the right-button menu.)

Expand and Collapse

Left-click on the triangle¹ to expand the object and see more information:

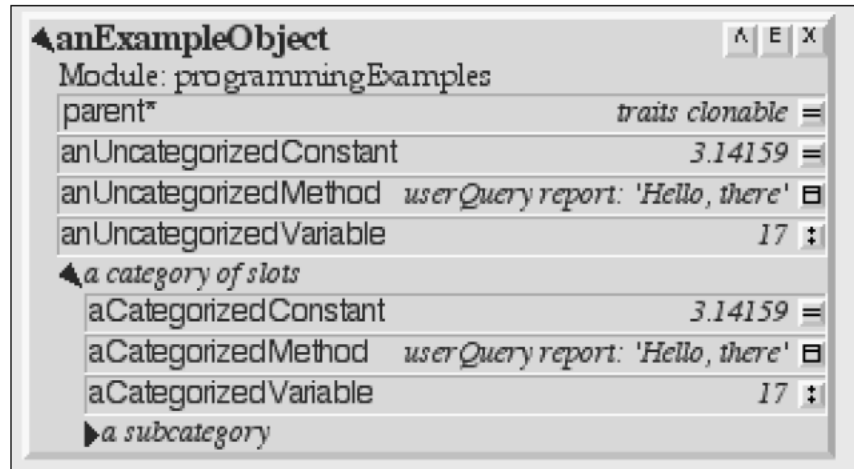


Now it shows a summary of modules containing the slots in this object (just `programmingExamples` here), four slots, and a category containing more slots, although those slots are not shown yet.

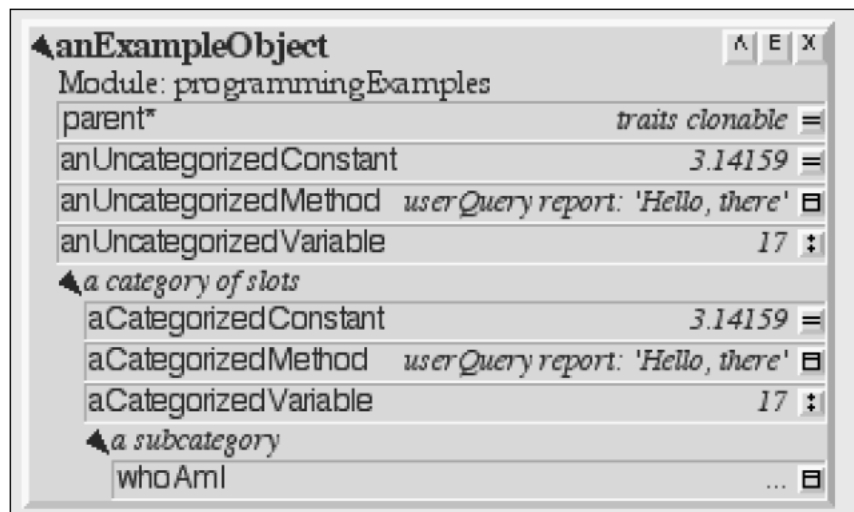
¹ Double-clicking on the triangle will expand (or contract) all levels instead of just a single level.

Categories




Clicking the top triangle now would collapse this object outliner, but instead look inside the category by clicking its triangle:



And, one more click expands the subcategory:

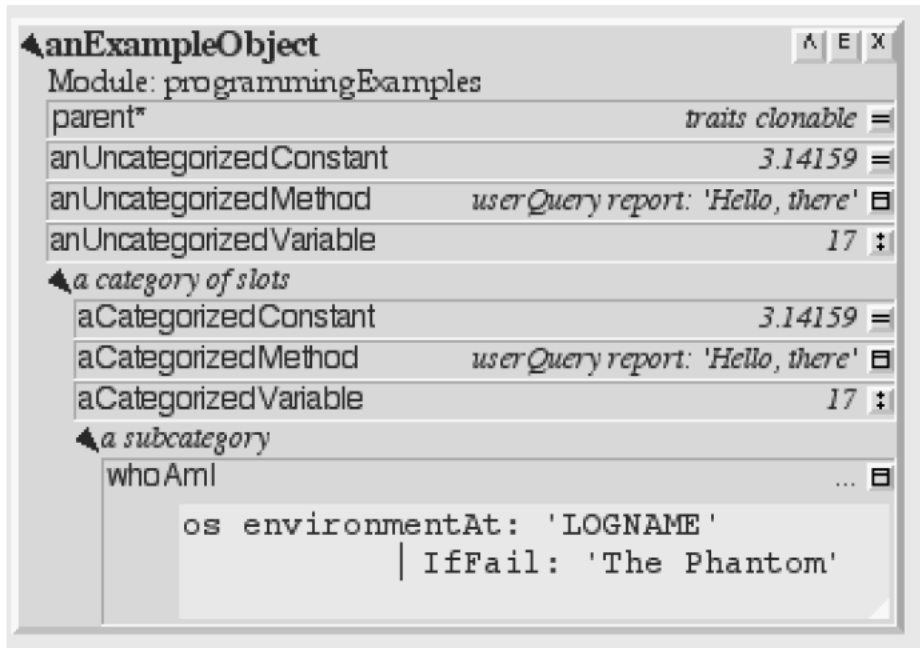


Slots

The little icons on the right edges of the slots reveal the type of slot:  for a method slot (a slot containing a method),  for a constant slot (a slot containing a data object), and  for an assignable slot (a pair of slots containing a data object and the assignment primitive). In order to save space, the data slot and its corresponding assignment slot are lumped together. (In other words in addition to the visible slot named aCategorizedVariable containing 17, there is another, *invisible* slot named aCategorizedVariable: containing the assignment primitive.)

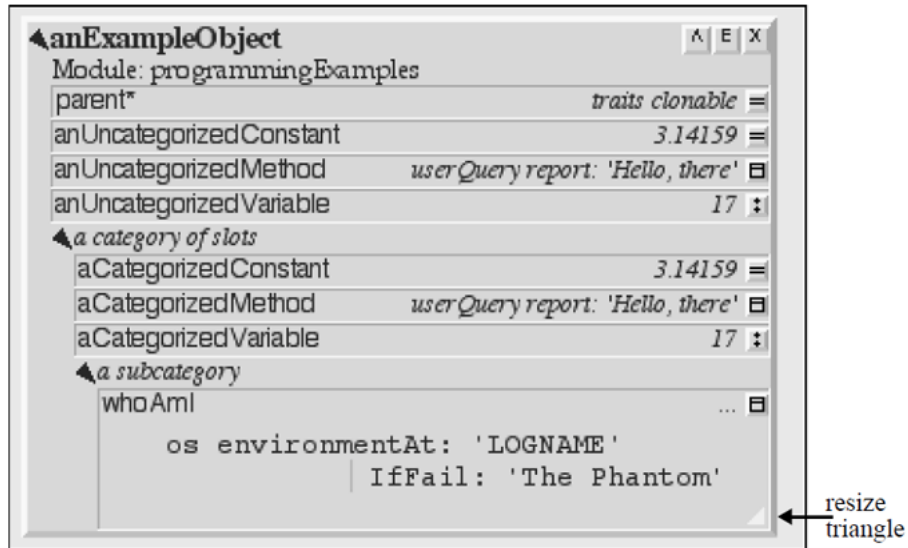
To look at the object contained in a data (constant or assignable) slot, just click on its icon. But if the slot is a method, clicking its icon opens up a text editor on its source. For example, clicking on the icon at the right of the whoAmI box

opens a text editor displaying its source (and typing control-L widens the object to show all the text in the selected window):



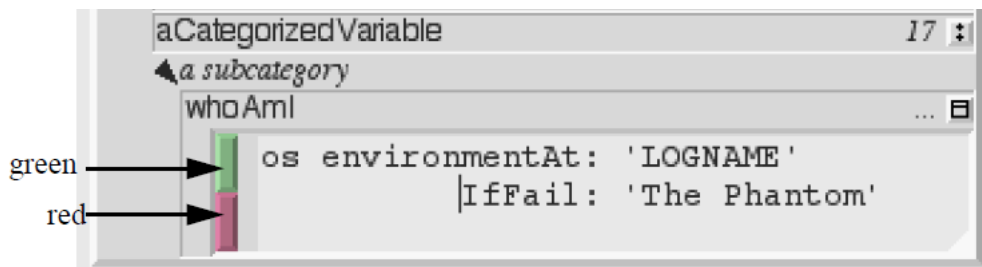
Text Editors

The background of the editor is lighter than the outliner as a whole, and this difference indicates that this editor is the current typing focus: no matter where the mouse is you can type into this editor. A left-click on another editor will select that one as the typing focus, and to indicate that it is no longer the focus, this editor's background will change to match the outliner:



The white triangle in the lower-right corner of the editor (which can barely be seen in the printout of this document) can be dragged to resize the editor.

Someone has done a poor job of indenting this method, so fix it by clicking to the left of the capital- I and deleting two spaces:



The red and green buttons that just appeared indicate the text has been changed; it no longer reflects the source code of the real method. Hitting the red button will cancel the changes, while hitting the green button will accept them and change the method:

Self text editors will honor the cursor arrow keys, the copy, paste, and cut Sun keys, and many emacs-style control characters:

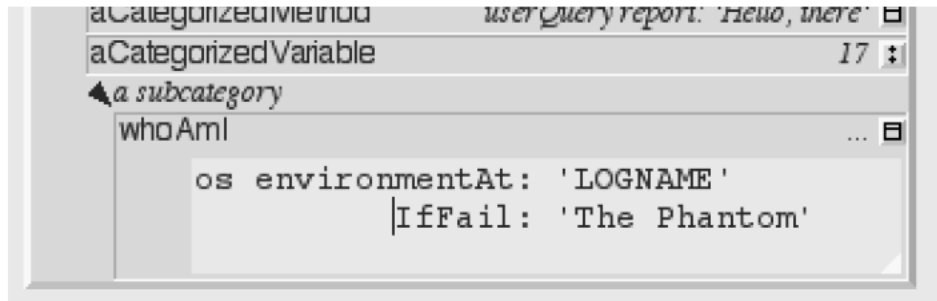


Table 6.1: Partial list of control characters in Self text editors

Character	Effect
control-a	Move to start of line.
control-b	Back one character.
control-d	Delete next character.
control-e	Go to end of line.
control-f	Forward one character.
control-k	Kill to end of line.
control-l	Expand the text editor to show the whole text.
control-n	Go to next line.
control-o	Open a new line after the cursor.
control-p	Go to previous line.
control-t	Transpose characters.
control-w	Erase previous word.
control-y	Yank text from past-buffer to editor.
delete, backspace, or control-h	Erase last character.
meta-return (command-return on Mac)	Accept.
escape (also command-period on Mac)	Cancel.
meta-s (command-s on Mac)	Save a snapshot.
meta-x (command-x on Mac)	Cut.
meta-c (command-c on Mac)	Copy.
meta-v (command-v on Mac)	Paste.
meta-d (command-d on Mac)	Dismiss morph containing typing focus.

Dismissing Objects

There are four separate ways of dismissing an outliner (or for that matter, anything) from the Self desktop:

- Object outliners: Push the “X” button at the top-right-hand corner.

- Drag it to the trash: left-drag on the outliner till the mouse is over the trash can, then release the mouse-button.



then

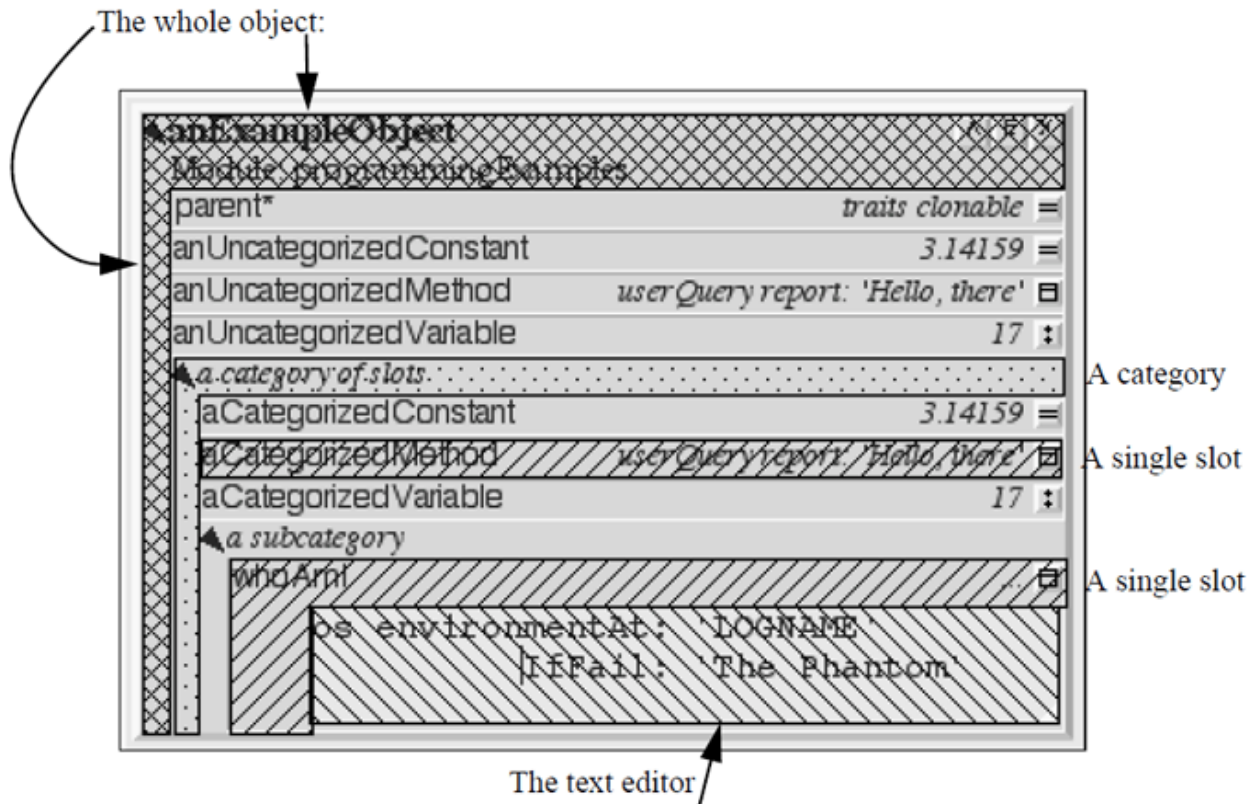
- Dismiss it via the right-button menu: hold down the right button over the outliner, move to the *Dismiss* button, then release.

- The Carpet Morph: start above (or below) and to the left (or to the right) of the outliner, over the background. Hold down the left button and sweep out an area that completely contains the outliner, then release the left button. The outliner should now be surrounded by a rectangle. Use the middle mouse button inside the rectangle to select *Dismiss*.

The last two methods, dismissing from the right-button menu, and marquee selection with the carpet morph, come in especially handy with things like buttons and menus because such morphs cannot be grabbed with the left-button.

6.2.2 Menus in the Outliner

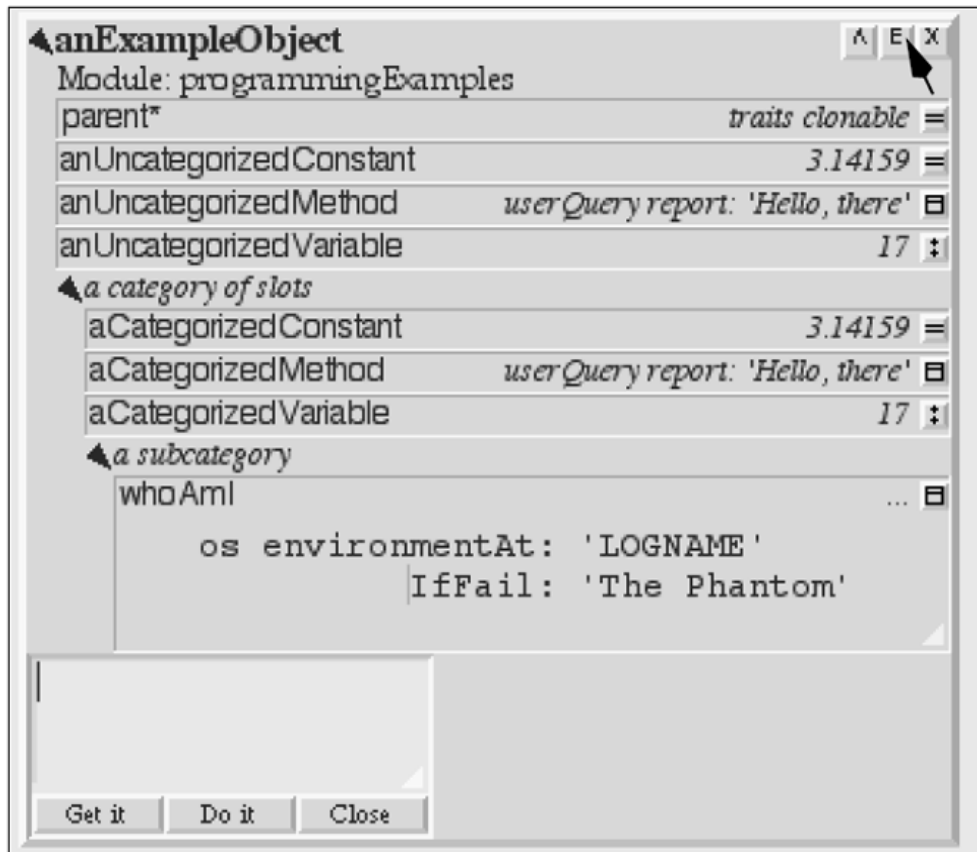
Many other operations are available on the outliner by using the middle-button menu on the part of the outliner to be affected. For example anExampleObject has many regions and here are some of them:



Click on the desired part of the object, be it object, category, slot, text editor, or annotation (annotations will be explained later).

The Evaluator

Try out the `whoAmI` method. Push the “E” button in the top-right of the outliner:

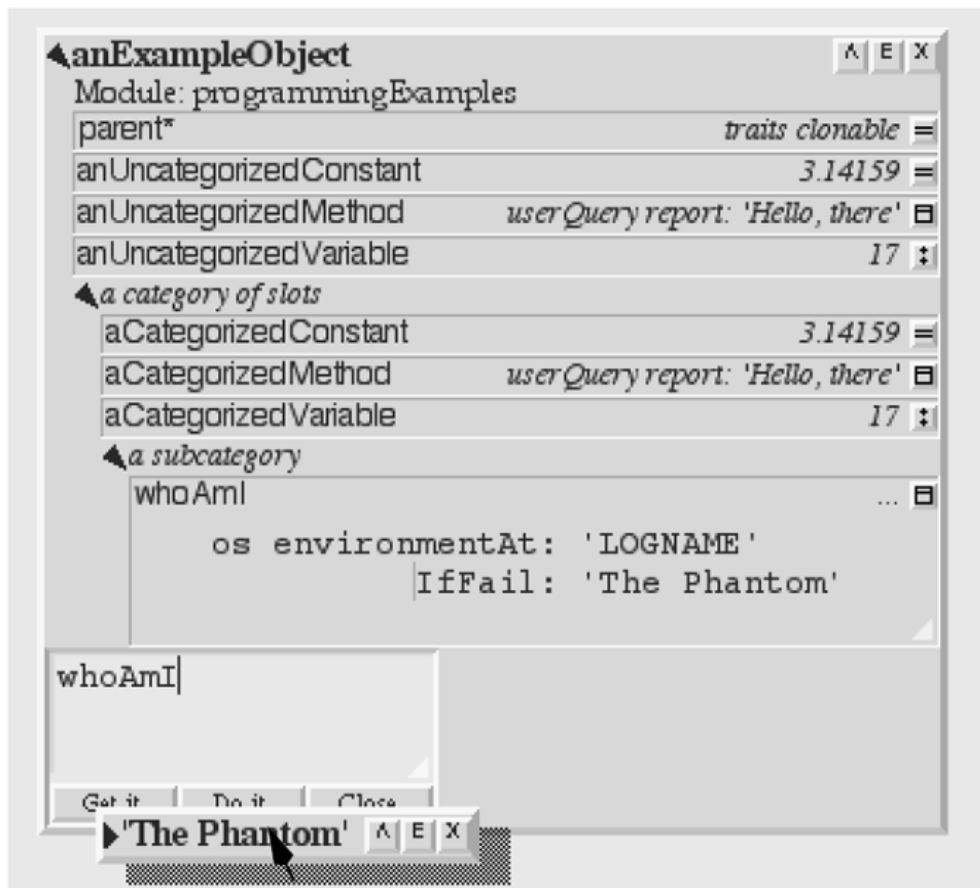


The receiver of any messages sent from an evaluator, or indeed any text editor (via *Do It* and *Get It* in the editor’s middle-button menu) in an object outliner is the object itself.² Type `whoAmI` into the evaluator and hit the *Get it* button (or select the *Get It* from the text editor menu), to send the message and get back the result:

Move the result³ out of the way and left-click to set it down.

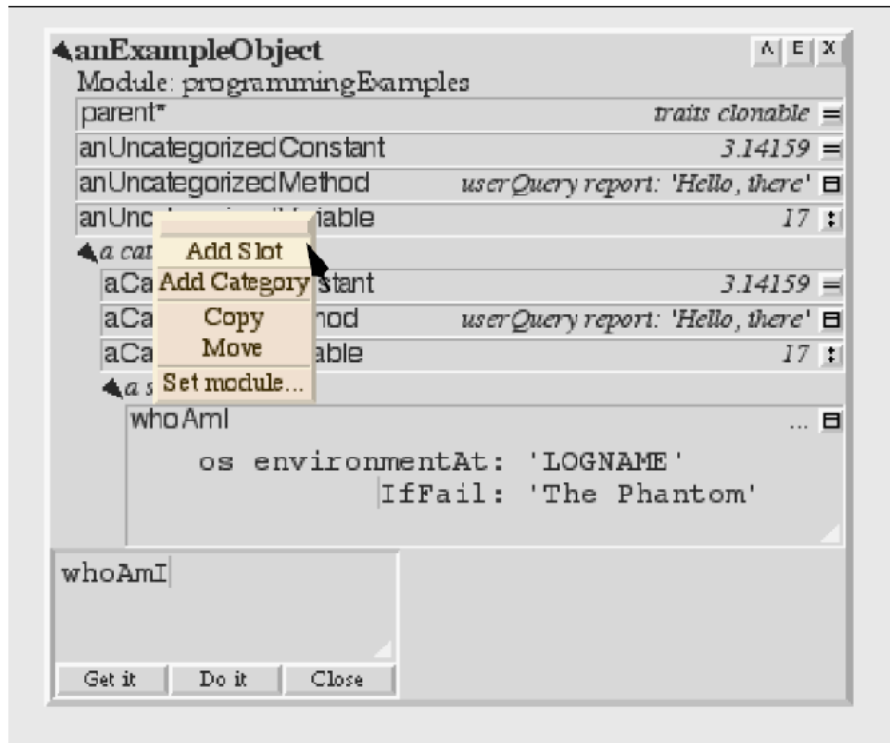
² However, in a stack frame in the debugger (described below), the receiver of a message is the same as the receiver for the stack frame.

³ I am revising this for Self 4.1 on my trusty Mac, and Self does not implement environment variables here.

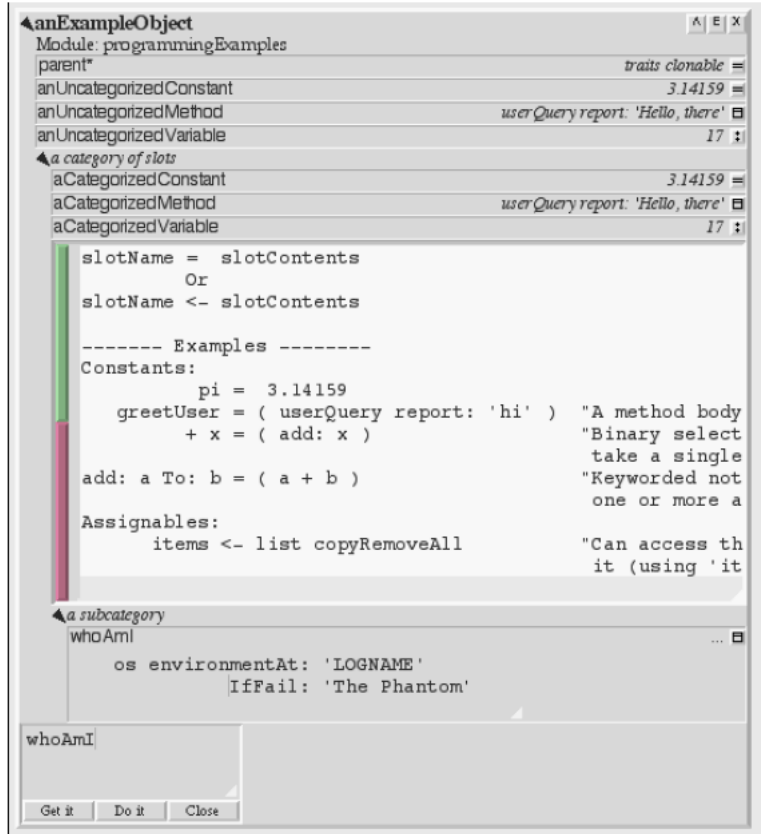


Adding a slot

Try one more change: adding a slot to the category “a category of slots.” Hold the cursor over the words a category of slots and select Add Slot from the middle-button menu.



After selecting Add Slot a space for a new slot will appear in the object:

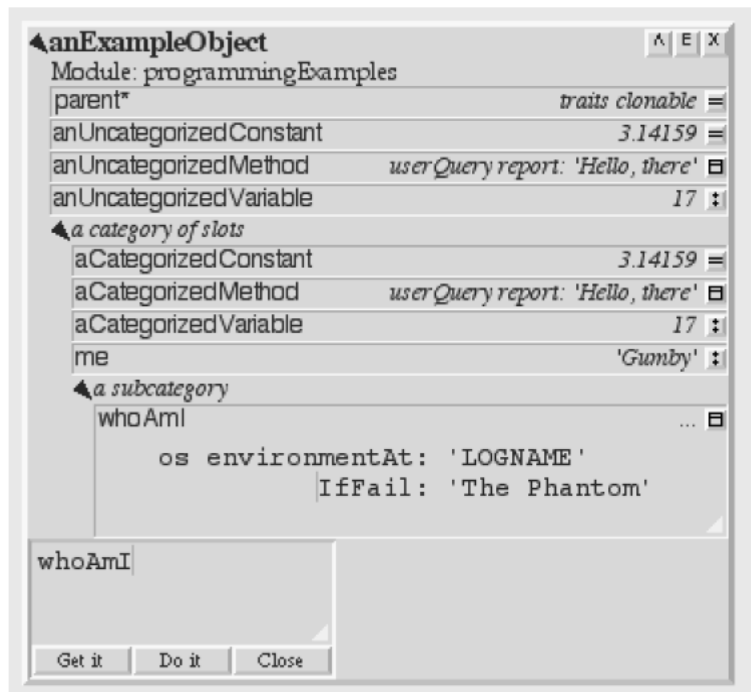
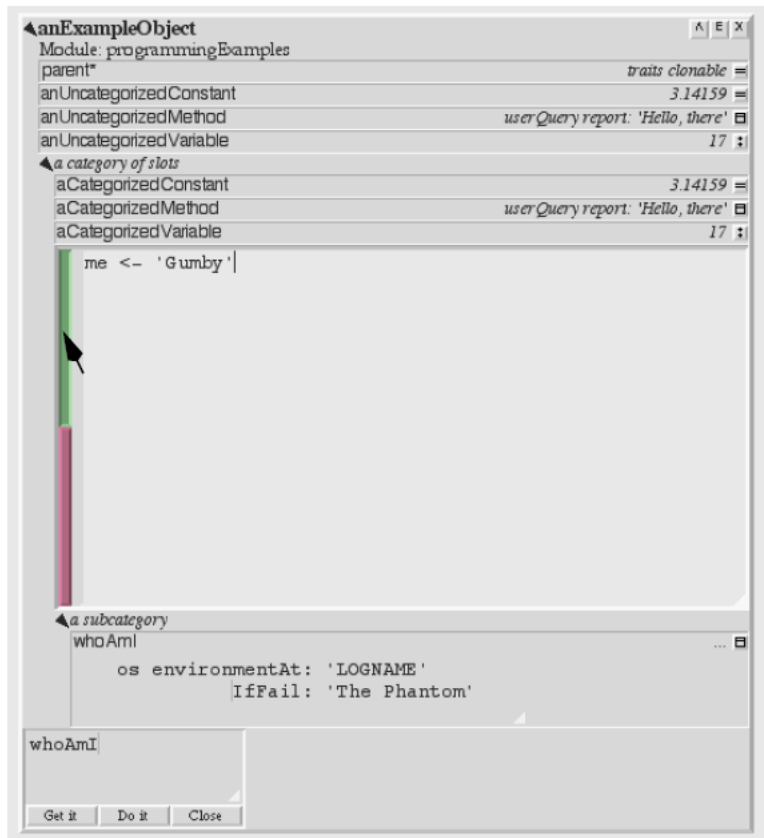


Each line shows the syntax for a different kind of slot. Create a simple variable by typing `me<- 'Gumby'`⁴ and hitting the green button to accept the change:

After releasing the green button, it stays down to let you know that it is still working. After a few seconds the slot appears:⁵

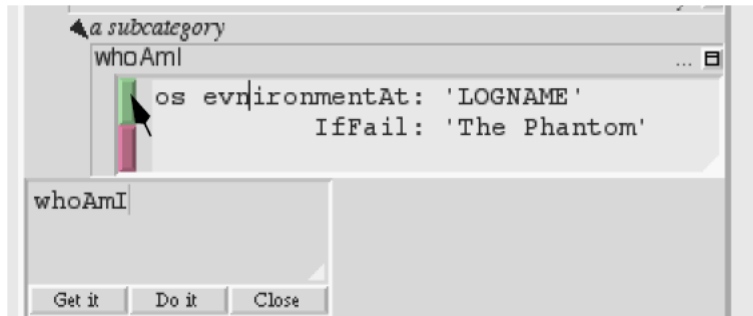
⁴ Since all that stuff in the text editor was initially selected, your typing conveniently replaced it all.

⁵ If you examine the slot's annotation (available via the slot menu) it will show that the system has guessed that the new slot (named "me") should be saved in the "programmingExamples" module, and that instead of saving its actual contents, the slot should just be initialized to the string 'Gumby'.

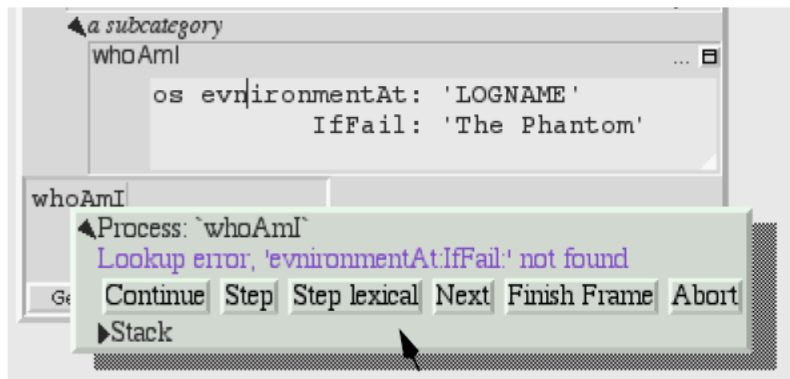


6.2.3 Debugger

Explore the Self debugger. Start by scrambling the send to `environmentAt:IfFail:` as if you had misspelled it.



Press the green button to accept the change, then hit the *Get it* button. This should break something! In fact, instead of the result of the message, a Self debugger will materialize:



The debugger has a label to indicate which process ran aground, a status indication shown in blue, some buttons for controlling the process, and a collapsed outliner for the stack. Expand the stack:

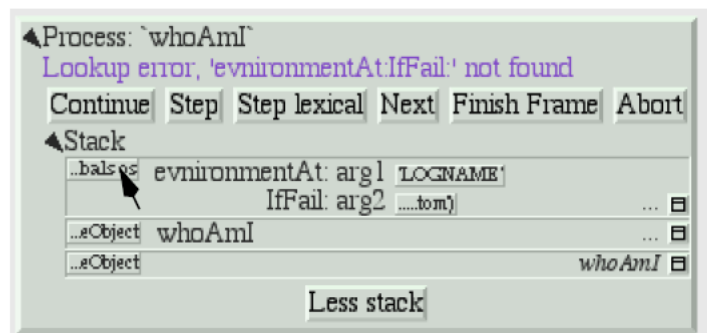


Fig. 6.1: Expanded stack.

The stack filters out uninteresting frames by default⁶. The debugger assumes that the first method you want to see is the one based on the text in the evaluator, and since the stack grows upwards this oldest frame appears at the bottom.

⁶ Since the Self compiler inlines calls automatically, Self code tends to be written in a highly-factored, deeply-nested style. Thus, the debugger filters out stack frames that seem to be unimportant. If it ever filters out the frame you need to see, there is a “Don’t filter frames” entry in the stack’s middle-button menu.

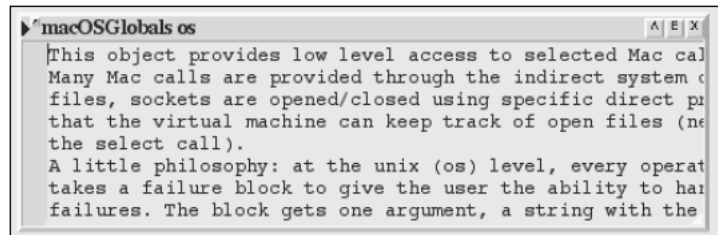
It has no method name, and contains the code `whoAmI`. That method called `whoAmI`, whose code is too long to show next to the slot button, and that method called `environmentAt:IfFail:` because we just sabotaged it! Of course there is no such method, but Self creates one dynamically to handle the error.

The little boxes represent the receiver and arguments of the methods on the stack. Get the receiver of the `environment...` message. Click on the box to the left of the word `environmentVariable:` (the one labelled “...bals os” if you are running on the Macintosh):

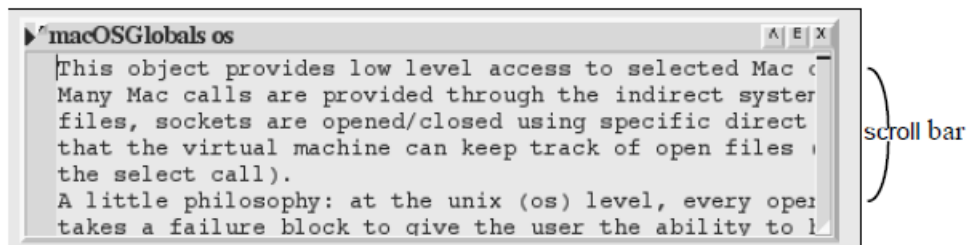


Fig. 6.2: Outliner with interface to the Macintosh.

This object represents the interface to the Macintosh operating system. The little button with the apostrophe in the top-left-hand corner indicates that this object has a comment. Push the button to show (or hide) the comment:

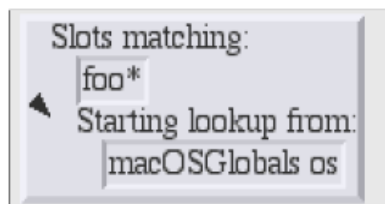


(To automatically resize the outliner to show all the text, press control-L.) To see one of Self’s scroll bars, grab the comment’s resize triangle (with the left-button) and move it up a bit:

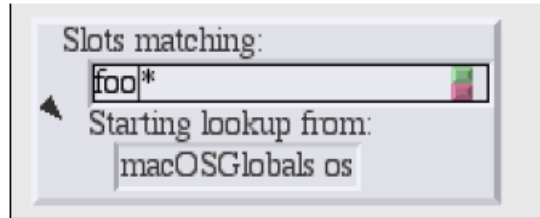


The affordance that appears on the right of the text is the scroll bar, and you can either drag on the little black line or just click in the bar to scroll the text up or down.

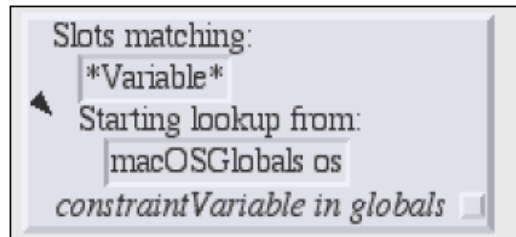
Push the little apostrophe button again to hide the comment. Now, you could expand this object to find the correct spelling of the “*environment...*” message. But instead, pretend that all you remember is that it has the word “Variable” in it somewhere. So, use a facility called *Find Slot* that takes a pattern and an object, and finds any matching slots in that object or its parents. Use the middlebutton on the outliner’s title (`macOSGlobals os`) to get the object menu and select *Find slot*:



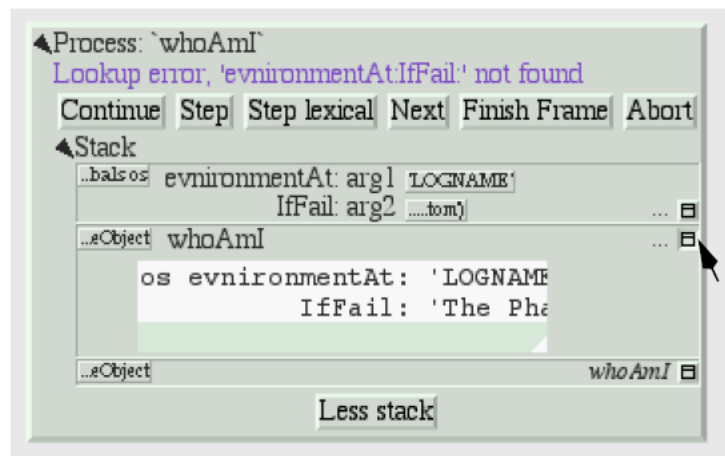
Double-click on the word “foo*” to select that field for editing. (The same trick works on slot names)



Since we are searching for a method with “Variable” in its name, backspace (the delete key on the Mac) three times⁷ to erase the “foo” type in “*Variable*”, hit the green button, and then hit the triangle to start the search (if you make a typing mistake, you can double click the text to make it editable again). The triangle will blink a bit while it is searching (one could do other things in the meantime during a long search), then the enumerator will show the match(es):



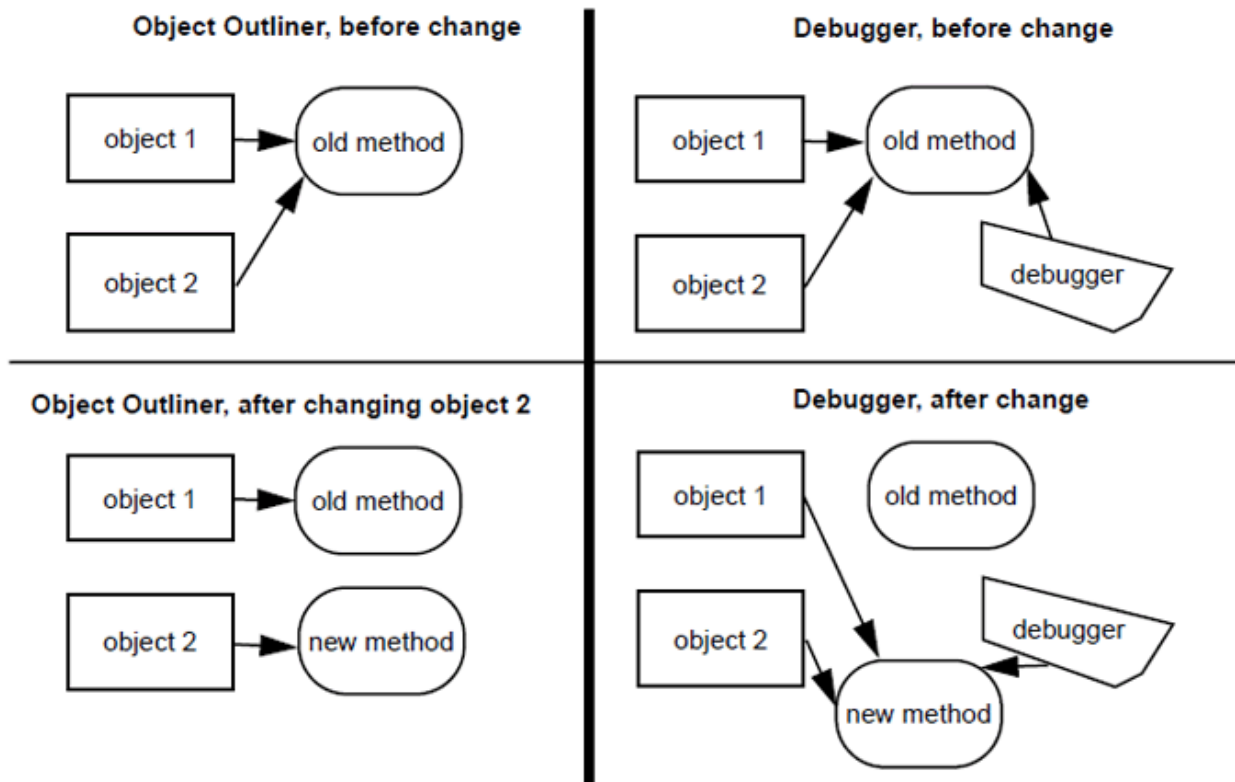
Clicking on the little square button(s) would show the exact method(s). But, for our purposes, just knowing the name is enough and now you have to fix it. So back to the debugger and click on the method button on the right in the whoAmI slot to expand the stack frame for the whoAmI method:



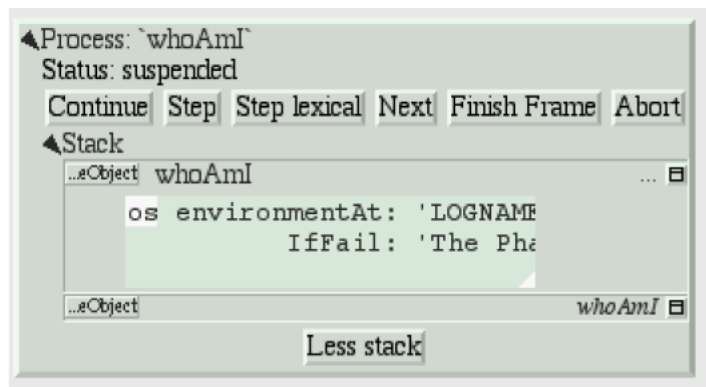
Now the debugger shows the source of the method, with the actual message being sent highlighted. (In this case it is just the whole thing.) One of the conveniences in the Self programming environment is that you do not have to go back to the original method to fix it, but can just fix it here (following the grand tradition of Lisp and Smalltalk programming environments). So use the left button to select the “vn” and type “nv” instead, then hit the green button to accept the change. The green button will stay in a bit longer because when a method is changed from the debugger, every slot pointing to that same method is made to feel the change—the method is changed in place (see the figure below). This feature lets you change a method in a clone and simultaneously affect the

⁷ One rough edge remaining in the Self user interface is the existence of two test editors, and this one does not implement multi-character selection, sigh. Or, you could type control-A to go the start, and control-K to delete the whole field, sigh.

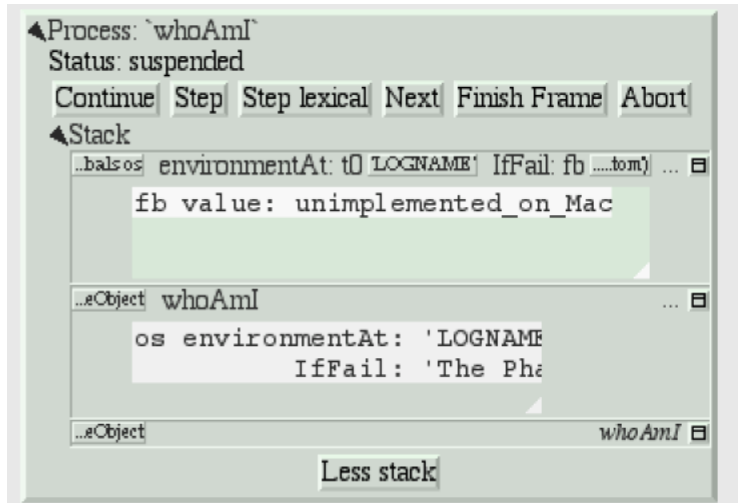
Changing a method in an object outliner vs. in a debugger



prototype, if you are putting your methods in prototypes instead of traits. Changing a method in an ordinary outliner would just affect that one object, even if other objects had been cloned from it. This rule avoids unintentional changes. The more global kind of change performed by the debugger takes a little longer. When it is accomplished, the red and green buttons will disappear:



Now `os` is highlighted to show that the process is about to send “os” to implicit-self. Try the `Step` button, which performs a single message send. After hitting the `Step` button twice (and a control- L to widen the debugger), the process will have entered the `environmentVariable: IfFail: method:`



This method is not too interesting (especially on the Macintosh), so leave the debugger by hitting `Continue` and letting the process finish.

Congratulations on making through the interactive tutorial. In the remainder of this manual, we will dive deeper into the programming environment for readers who want to write real programs in Self.

Here is more information on the debugger for future reference:

Table 6.2: The Debugger Buttons

What it says	What it does
Continue	Resumes running the process
Step	Perform one message send (skipping over trivial accesses and assignments); Steps into the called method.
Step Lexical	Execute messages until control returns to the same lexical method, or until this method exits. Very useful for methods with blocks.
Next	Performs a message send and any messages in the called method; Steps over the called method.
Finish Frame	Finishes running the topmost method.
Abort	Kills off the process and dismisses the debugger.

In addition to the buttons, each frame in the debugger has some items to control the process in its middle-button menu:

Table 6.3: Process control items in the activation middle-button menu

What it says	What it does
Step	Top frame: same as step button, not top frame: Finish any called methods.
Next	Same as next button.
Retry	Cut back the stack to this frame, then continue the process.
Revert	Cut back the stack to this frame.
Finish	Finish this frame.

6.2.4 Enumerators

In addition to the *Find Slot* enumerator, Self has other ways to find things:

Table 6.4: Enumerators

Name	Function
Implementors	Finds all the slots with a given name.
Implementors of :	Finds all the slots with the given name that take an argument (for read/write slots only).
Senders	Finds all the methods that send a message with a given name.
Senders of :	Finds all the methods that send the corresponding assignment message (read/write slots only).
Senders in family (Senders of : in family)	Finds all the methods in this object, its ancestors, and descendants that send a message with a given name (or the corresponding assignment message).
Find Slot	Starting from a designated object, finds all slots in that object and its ancestors whose name matches a given pattern. Case is ignored, “?” matches any character, “*” matches any series of zero or more characters. Also comes in “of :” and “in family” flavors.
Methods Containing	Finds methods containing the specified string. Similar to <code>grep</code> without wildcards.
Copied-down Children	Finds objects copied-down (see below) from this one.
References	Finds slots that contain references to the selected object.
Slots in Module	On the module menu (see below); shows all slots in a given module.
Added or Changed Slots in Module	On the module menu (see below); shows all slots added or changed in the module since it was filed out.
Removed Slots in Module	On the module menu (see below); shows the names of the slots removed from the module since it was last filed out.
Expatriate Slots	On the changed module menu (see below); shows all slots in filed-out objects that do not themselves specify a module. These slots will not be filed out.

The `copy-down` and `module` enumerators will be covered later.

The other enumerators can be summoned from several places: the outliner menu, the slot menu, and the text editor menu. As a shortcut, selecting a whole expression in the text editor and then asking for an enumerator will bring up the enumerator to search for the outermost message send in the expression. So if you select the following expression: `aSet findFirst: elem IfPresent: [snort] IfAbsent: [sludge]` and choose implementors from the text editor menu, you will get an Implementors enumerator ready to search for `findFirst:IfPresent:IfAbsent:..` Of course, you can always change the search target by double-clicking and editing the text. The text editors also implement a host of handy double-clicking shortcuts.

Finally there is one last detail about enumerations: many contain a check-box to choose *Wellknown only*. This is always checked by default to speed things up. When checked, only wellknown (i.e. filed-out, see below) objects are searched, which is much faster.

6.3 Hacking Objects

Hacking—the discipline of making fine furniture from trees using an axe.

In going through this document, you have already added a slot and edited methods in both object outliners and debuggers. In addition Self has many other ways to change an object:

Table 6.5: Ways to change an object

Ways to change an object	How	Why
Removing, Moving, Copying Categories		
Removing a category.	“Move” in category middle menu, then drag the category to the background or the trash can.	Removing a category.
Add slot or category to object or category.	“Add Category” in object or category middle menu, then type in the new category name, then hit green button to accept.	Adding a new category.
Moving a category.	“Move” in category middle menu, then drag to another object.	Copying a category.
Copying a category.	“Copy” in category or category middle menu, then drag the category to another object.	Copying a category.
Removing, Adding, Moving, Copying Slots		
Removing a slot.	“Move” in slot middle menu, then drag the slot to the background or the trash can.	Removing a slot.
Add slot to object or category.	“Add Slot” in object or category middle menu, then type in the new slot name, “=” or “<-”, and contents of slot (or just name alone for variable slot containing nil), then hit green button to accept.	Adding adding a new.
Moving a slot.	“Move” in slot middle menu, then drag to another object.	Moving a slot.
Copying a slot.	“Copy” in slot or category middle menu, then drag the slot to another object.	Copying a slot.
Adding a Comment		
Add a comment to an object or slot.	“Show Comment” in the object or slot middle menu to open up a comment text editor, then typing in the comment, then hit the green button to accept it. If an object or slot already has a comment, it can be shown/hidden by hitting the small button labeled with a single quote.	To amuse and intrigue those who follow.

Changing a slot		
Edit a slot.	“Edit” on a slot middle-button menu, then make any changes in the text editor, then hit green button to accept changes.	To change the contents of a constant data slot, or to change contents and set initial value at same time, or to change a slot from data to method or from constant to variable.
Edit slot name or its argument names.	Double-click on the name of the slot, wait for red and green buttons to appear on the right of the name, edit the name, then hit the green button.	To change a slot’s name or the names of its arguments.
Change a method in a slot.	Click on the method icon button on the right of the slot to open a text editor on the method. Make the changes, then click on the green button to accept them.	To fix a bug in a method.
Change the visibility of a slot.	On the slot’s middle menu choose “Make Public,” “Make Private,” or “Make Undeclared.”	The Self interface uses bold, normal, and sans-serif fonts to indicate public, private, and unspecified slots. This distinction carries no semantics, but serves to record the programmer’s intentions.
Annotating an Object		
Change creator annotation of an object.	“Show Annotation” in object middle menu to expose object annotation information, then click on creator path field and typing in desired creator path, then hit green button to accept annotation.	Setting creator path tells transporter which slot “owns” this object, and tells environment what to name the object.
Set creator of contents of a slot to that slot.	“Set Creator” in slot middle menu.	See above.
Change copy-down information.	“Show Annotation” in object middle menu to expose object annotation information, then click on copy-down-parent field and type in desired copy-down-path, copy-down selector and slots to omit) then hit green button to accept annotation.	Simulates subclassing by allowing an object to contain copies of the slots in another object. When copy-down-parent has slots added/changed/removed, the change propagates to the copied-down children.
Change the object’s “isComplete” flag.	“Show Annotation” in object middle menu to expose object annotation information, then push one of the isComplete radio buttons, then hit green button to accept the annotation change.	After building a new prototype, set isComplete to get the environment to show its printString, and to get the transporter to use its storeString.

Annotate a slot		
Set the module membership of a slot, the slots in a category, or the slots in an object.	Select “Set Module” from the middle menu of a slot, category, or object, then (for object or category) indicate which slots you want to change by choosing which module they currently belong to, finally select a new module to put the slots in.	To ensure that slots are filled out in the correct source file.
Type in or examine the module for a single slot.	“Show Annotation” on the slot middle menu to expose the annotation, then click on the module editor, type in the module name, then click the green accept button.	Save as above.
Change slot initial contents.	“Show Annotation” on the slot middle menu to expose the annotation, then click on the “Follow Slot” button, or type the desired initial value expression into the “Initial Contents” editor, then hit the green accept button.	To have the transporter record the current contents of a slot, choose “Follow Slot.” To have it ignore the current value and just record a given expression for the slot’s initial value use the “Initial Contents” option.

6.4 The Transporter

The transporter has been built in order to move programs from one world of objects to another. You can ignore it as long as you work with just one snapshot. However, if you want to give your program to someone else, or save it as source, or read it in to a newer snapshot, you will need to learn about the transporter.

6.4.1 The Traditional Schism between Program and Data

What is a program? In most systems it is a piece of text, although in more advanced environments it may have structure. It is a description that can be used to create an activity, a running program, that can then operate on data. In the conventional view:

Table 6.6: The Schism between Program and Data

	Program	Data
Who can change it	The programmer	The user
When can it change	At programming time	At execution time
How is it changed	With a text editor	By running a program

This model grew up in an era where computers were too small to host both compilers and applications at the same time. Although it has some virtues it makes other operations very hard: it is hard to include data, such as hand-drawn icons, directly into a program, and it is hard to write applications whose data domain is really programs.

6.4.2 Data = Program

For Self, we have gone a different way, following in the footsteps of Smalltalk and Lisp:

A Self program consists of live objects.

Self has no edit/run mode. To change an object, you do not retreat to a source file, or even to a class, you just change the object itself. This immediacy and concreteness lessens the cognitive burden on the programmer, smooths the learning curve, and hastens gratification.

However, this stance creates a big problem the moment you need to move a program from one world of objects to another; it is very hard to pin down what to do. For example, suppose an object contains a slot with 1024 in it. Should that value be copied literally? Perhaps it is the result of some computation (such as the width of the current screen) and should be recomputed instead. There simply is not enough information in a Self object to extract programs from Snapshots.

6.4.3 Changes vs. Pieces

Earlier in the project we considered constructing a calculus of changes that could be used to represent programs, and then moving programs by reapplying the changes to the new snapshot. But, we had enough on our plate and rejected this approach as too ambitious to tackle without a dedicated graduate student.

Instead, we decided to represent programs as pieces that could be filed out of a snapshot and filed in to another. To allow us to merge changes to the same program, we decided to represent its pieces as Unix source files amenable to RCS. The Self Transporter was built to save programs as source files.

6.4.4 Objects vs. Slots

But what is a program? Although a new program frequently involves creating new objects, it also can mean added slots to existing objects. For example, a program to find palindromes might add a slot to `traits string` called `isPalindrome`. So we decided to refine the granularity of the Transporter to the slot level; each slot has an annotation⁸ (its module) which gives the name of the source file containing that slot. This hair- or rather object-splitting implies that one object may be built incrementally as the result of reading several files, and so the transporter endeavors to keep the order that the files are read in as independent as possible. Since each object can possess slots in different modules, the outliner shows a summary of the modules of an object, sorted by frequency.

Turned around, a module can be viewed as a collection of slots, plus some other information: each module also includes a directory, a list of submodules to be read in whenever it is read, and `postFileIn` method to be run whenever the module is read. These data allow modules to be organized hierarchically by subsystem, for example the `allUI2` module includes all the modules in the `ui2` system.

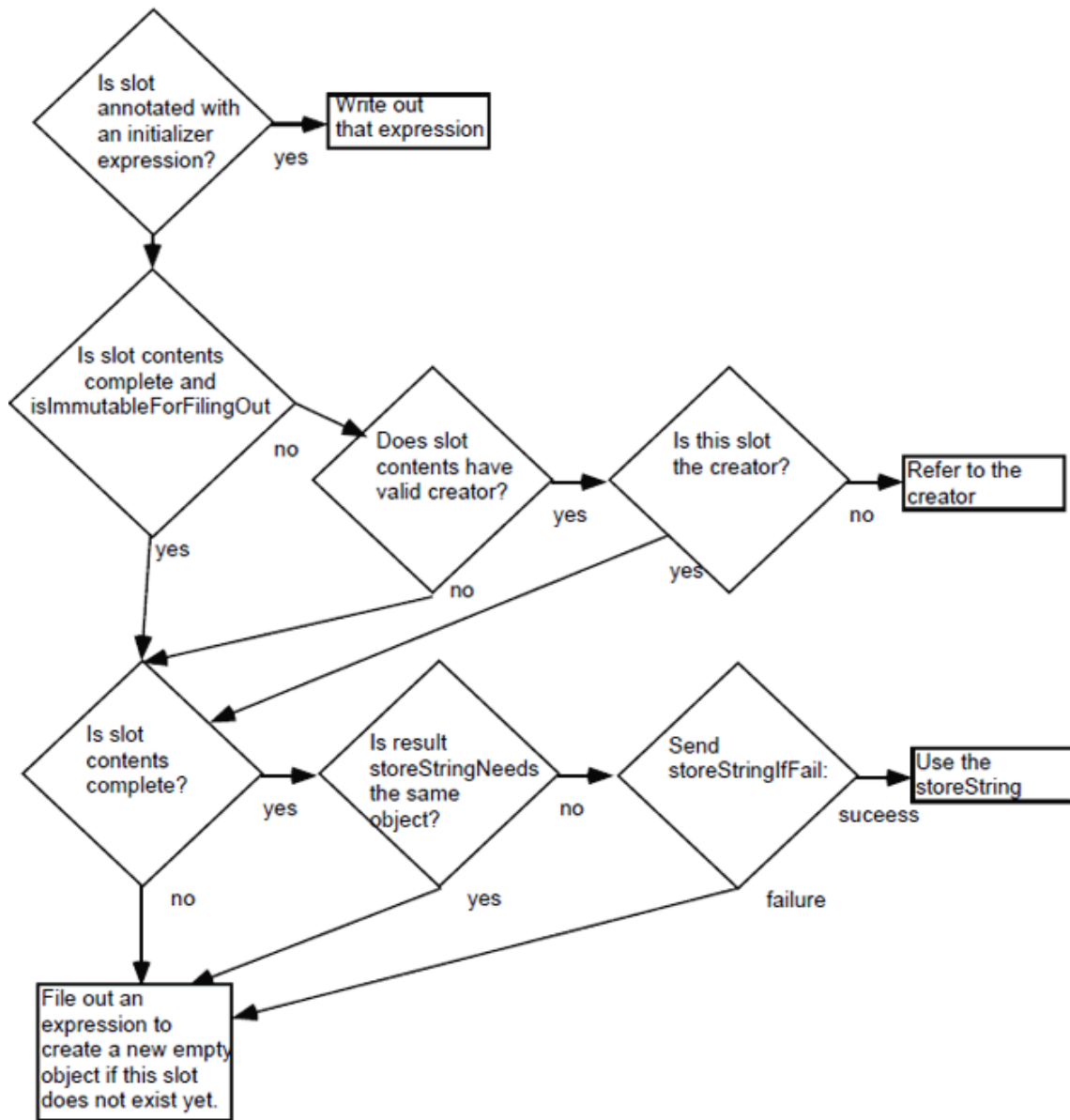
Now here comes the nice part: the Self environment incrementally maintains a mapping from modules to slots, and a list of changed modules, which can be obtained from the background menu. When you make a change the appropriate module will be added to the list, and can be written as a source file by clicking its ‘W’ button. The middle-button menu on the changed modules and individual modules contains a host of useful entries for understanding what has been changed.

6.4.5 What to Save for the Contents of a Slot

At this point, the reader may be thinking “*So modules know which slots they include, but how do they know which objects to include?*” After all, when the transporter saves a slot in a file what can it put for the contents of the slot? Here is where the transporter runs smack into the problem of not enough information, and a variety of means have to be used. As shown in the flowchart below:

⁸ The Self Virtual Machine provides for annotations on slots or whole objects. While the annotations do not influence program execution, they can be accessed and modified by Self’s reflective facility, mirrors. Annotations are used to hold many things, including comments on objects and slots.

How the transporter files out objects



- Sometimes the programmer does not want to store the actual contents of a slot, but instead wants to store an initialization expression. This intention is captured with another annotation on a slot: each slot can either be annotated *Follow Slot* or *Initialize To Expression*. In the latter case, an initializer is also supplied.
- Even though the transporter is supposed to follow the slot, it may contain an object that is created by another slot. For example, the `parent` slot in a `point` should just refer to `traits point` rather than recreating the `traits` object. This information is captured by a `Creator` annotation on each object that gives the path from the `lobby` to the slot intended to create the object. In this case, the transporter just files out a reference to the object's creator, cleverly enough so that the actual creator slot does not need to have been already filed in. On the other hand, if an object is immutable, its identity is not important. If an object is annotated as `isComplete`⁹ the transporter sends it `isImmutableForFilingOut` and if that message returns `true`, the transporter never files out a reference. For example, integers would answer `true` to this message.
- If the contents of the slot is a simple (usually immutable) object like `17`, `3@4`, or `'foo'` (the string) the trans-

⁹ `isComplete` is used by the environment to decide when it is safe to send messages like `printString`.

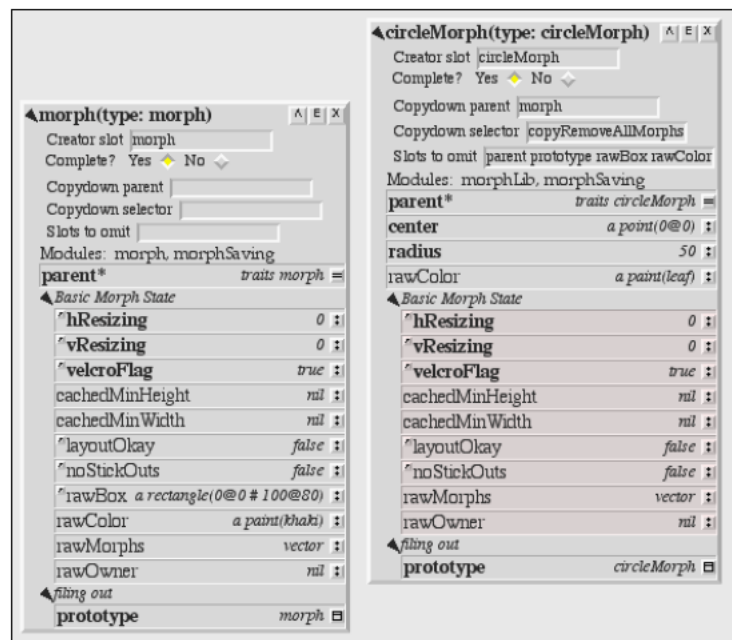
porter should just ask the object for a string to store. It does this by checking to see if the object is annotated as `isComplete` to see if it is safe to send the object messages, checks to see if this object is itself needed for the string (it would be a mistake to file out the prototypical point as `0@0`, because the `x` slot would never be defined), then asks the object for a store string. To see if the object must itself be filed out, it sends `storeStringNeeds` and if this message does not return the object itself it sends `storeStringIfFail:`. If this succeeds, the transporter can save a data-type specific string for the object. This fairly elaborate mechanism allows programmers to add new kinds of objects that transport out with type-specific creation strings.

- Finally, if it can do nothing else, the transporter creates a new object for the contents of the slot. The object is created in a clever way so that a file that adds slots to an object can be read before the file that officially creates the object without loss of information.

Filing out objects is too complicated, and over the past two years we have repeatedly tried simpler schemes. However, all of the capabilities in the current scheme seem to be essential in some case. This issue remains as a question for future work.

6.4.6 Copy Down

Because Self eschews classes and because the current compiler cannot optimize dynamic inheritance, it is necessary to copy-down slots when refining an object. For example, the prototypical morph object contains many slots that every morph should have, and some mechanism is needed to ensure that their presence is propagated down to more specialized morphs like the `circleMorph`. In a class-based language, this need is met by a rule ensuring that subclasses include any instance variables defined in their superclasses. In Self, this inheritance of structure is separated from the inheritance of information performed by the normal hierarchy of parent slots. Instead of including a facility for inheriting structure in the language, Self implements a facility in the environment, called “copy-down.” An object’s annotation can contain a copy-down parent, copy-down selector, and set of slots to omit. The copy-down parent is sent the message given by the copy-down selector, and (except for the slots-to-omit), the slots in the result are added to the object. Copied-down slots are shown in pink in the outliner. For example, here are the prototypical morph and the prototypical `circleMorph`:



The *Basic Morph State* category of slots has been copied from those in `morph` by first copying the `morph` and removing all its submorphs (i.e. by sending it `copyRemoveAllMorphs`) and then copying the resultant slots, omitting `parent`, `prototype`, `rawBox` and `rawColor`. The first three of these slots were omitted because their contents

had to be different; copied-down slots are copied, they cannot be specially initialized in Self. The omitted slot `rawBox` is more interesting; circle morphs do not need this slot at all and so omit it. Most other object-oriented programming systems would not allow a subclass to avoid inheriting an instance variable.

The Self programming environment uses the copy-down information to allow the programmer to use a class-based style when appropriate. For example, if the programmer adds a slot to morph the environment will offer to add it to `circleMorph`, too. If the programmer should use a text editor to edit the definition of morph, the `circleMorph` object will be changed after rereading both object's text files. The least convenient aspect of using copy-downs is that to do the moral equivalent of creating a subclass, the programmer has to create two objects: a new traits object, and a new prototype, and then set the object annotation of the new prototype. Perhaps someday there will be a button to do this, or perhaps other styles of programming will emerge.

6.4.7 Trees

By default, the transporter writes out Self modules out to a tree rooted in the current working directory, or the 'objects' subdirectory of the directory given to the VM in the shell environment variable `SELFWORKING_DIR`.

However Self modules have a slot 'tree' which can take a name of a tree. If the name of the tree is not an empty string, then the module writer will look up a directory in the dictionary found at `modules init treeDictionary`.

This allows the developer to maintain several separate trees. For example:

```
modules init
  registerTree: 'org_selflanguage_webserver'
              At: 'path/to/parent-folder'.

bootstrap read: 'webserver'
              InTree: 'org_selflanguage_webserver'.
```

Important considerations: module names are globally unique (that is, two modules called 'webserver' in different trees are considered the same module and will overwrite each other). The tree name itself should also be globally unique - that is it is not possible to have two trees with the same name in a single Self world.

The advantages of this over a simple symbolic link to a separate filesystem tree is we can do overlays - if you want special string behaviour, then put it in your tree in `my_tree/core/string.self` and it will override as expected.

Modules that import subparts will import them from the same tree by default.

6.4.8 Versioning

Each transporter module has a slot named `revision` containing a string version number. It is recommended that you use Semantic Versioning¹⁰ so that the version of a module can be tested as follows:

```
modules string version >= (modules init moduleVersion copyOn: '1.0.0')
  ifFalse: [log warning: 'Old string version']
```

This test could be placed in the `preFileIn` slot of your module to ensure a sane file in environment before the rest of the file is read.

This concludes a brief tour of the Self programming environment. Although we strove for simplicity in the design of Self, its programming environment includes a fair amount of functionality which may take a while to learn. We hope that you find the investment worth the reward.

¹⁰ See <http://semver.org> for a specification. In essence, versions are of the form "3.2.1-alpha6".

MORPHIC: THE SELF USER INTERFACE FRAMEWORK

7.1 Overview

Morphic is a user interface framework that supports composable graphical objects, along with the machinery required to display and animate these objects, handle user inputs, and manage underlying system resources such as X displays, fonts, and color maps. A primary goal of morphic is to make it easy to construct and edit interactive graphical objects, both by direct manipulation and from within programs.

A Morphic window shows just one portion of a large, two-dimensional world. The window can be panned around in this world, allowing different areas to be used for different activities. Multiple users can be active in this world simultaneously, working either in the same area (for collaborative work) or in disjoint areas (for independent work). Every user can see the screen boundaries, cursors, and actions of the other users. The sharing is implemented via the X protocol, so the other users can be physically remote as long as there is sufficient network bandwidth to support the X traffic. (Preliminary tests suggest that 30-50 KBytes/sec is required between the host machine and each remote user. The system has been tested with up to five users active simultaneously.) A given user can have several Morphic windows open, either on the same world or on different worlds, and can drag objects among these windows (assuming they all belong to the same Self Unix process).

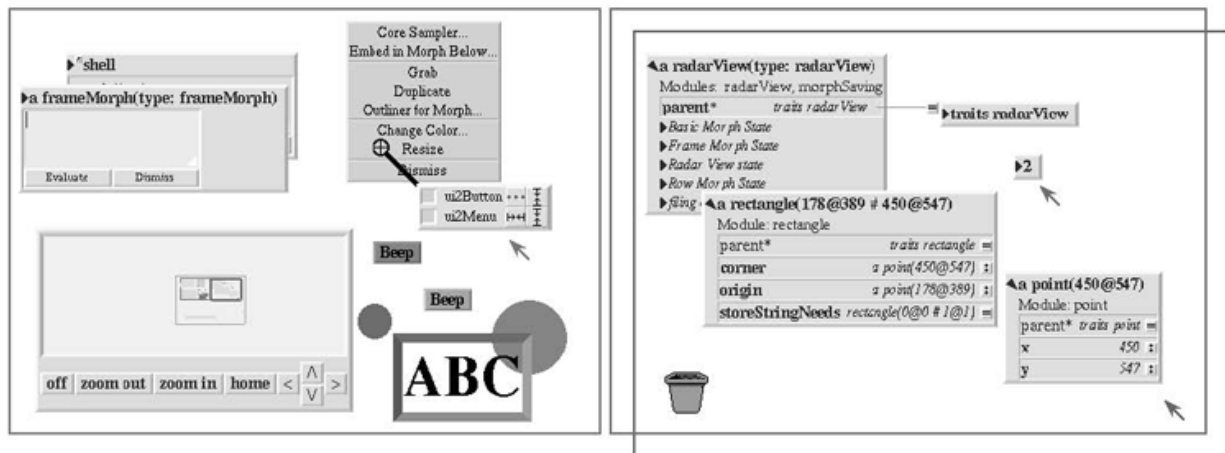


Fig. 7.1: Three users working in the same space. Two of the other users are collaborating, so they have made their windows overlap (right). The third user is working independently in a separate area (left). The `radarView` in the third user's area shows the surrounding vicinity in miniature, allowing offscreen objects and the screen boundaries of other users to be seen. The `radarView` is updated open enough to see where activity is occurring.

The central abstraction of morphic is the graphical object or *morph* (from the Greek for “shape” or “form”). A morph has a visual representation that can be picked up and moved. In addition, a morph may:

1. perform actions in response to user inputs,
2. perform an action when a morph is dropped onto it or when it is dropped onto another morph,
3. perform an action at regular intervals, and
4. control the placement and size of its submorphs.

Any morph can have component morphs (called *submorphs*). A morph with submorphs is called a *composite* morph. A composite morph is treated as a unit; moving, copying, or deleting a composite morph causes all its submorphs to be moved, copied, or deleted as well. By convention, all morphs are visible; morphic does not use invisible structural morphs for aggregation. This means that if a composite morph is disassembled, all its component morphs can be seen and manipulated.

The remainder of this document discusses the graphics interface that morphs use to draw themselves, the structure of composite morphs, how to create new kinds of morphs, and how to change a morph's behavior with respect to user inputs, drag-and-drop, and animation.

7.2 Composite Morphs

A *composite* morph is a morph that contains other morphs known as *submorphs*. Note that the terms “composite” and “submorph” refer to roles that can be played by any morph. Submorphs can be added to any kind of morph—even morphs such as `circleMorphs` or `labelMorphs` that are atomic in some other systems. Copying, deleting, moving, drawing, and layout operations are applied to the composite morph as a whole.

The structure of a composite morph forms a tree. When morph B is a submorph of morph A, B's *owner* is A and B appears in A's *submorphs* list. A morph can only be a submorph of at most one morph at a time, so its owner is a single value, not a collection. A window containing a collection of morphs is itself just a morph known as a `worldMorph`. Each user's cursor is represented by a `handMorph`. Grabbing a morph with the mouse is implemented by removing the target morph from the `worldMorph` and adding it to the `handMorph`. Dropping a morph when the mouse is released is implemented by reversing this process. A `handMorph` is itself a submorph of its world. The message `root` can be sent to a morph to get the top-most owner of a composite morph (stopping just short of the `worldMorph` or `handMorph` that contains it). If a morph is not a submorph of any morph, its owner is `nil`.

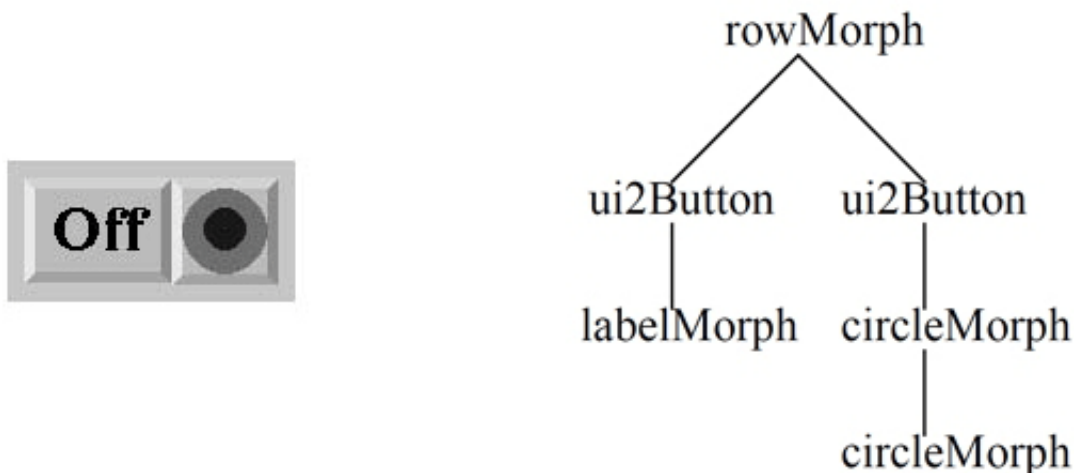


Fig. 7.2: A composite morph consisting of a row with two buttons. Each button has submorph to indicate its function; one is a piece of text, the other is an icon consisting of two concentric circles. The diagram on the right shows its submorph structure.

A morph can be made a submorph of some other morph using the `addMorph:` operation. This operation updates

both the owner slot of the submorph and the submorphs list of the owner to reflect the desired configuration. For example, adding morph B to morph A adds B to A submorph list, removes B from its old owner (if any), and sets B's owner to A. The `addMorph:` operation also updates the layout of both B's old and new owners. The global position of a morph is held invariant by `addMorph:` (although some morphs may perform an automatic layout as a side-effect of `addMorph:`, immediately changing the position of the newly-added morph).

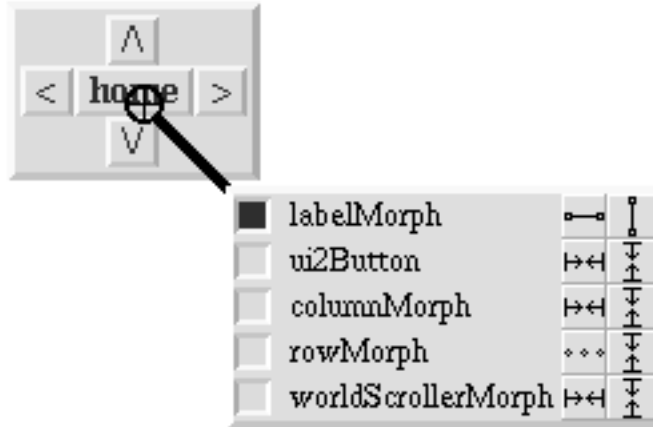


Fig. 7.3: Using the core sampler to probe the submorphs at the point below its cross-hairs.

The programming environment includes a tool called the *core sampler* (available via the right-button menu of any morph) that can be used to explore the submorph structure of a composite morph. The core sampler shows the set of submorphs below a given point, the way a core sample of rock allows geologists to study the strata of rock at a given point. The core sampler allows one to use the middle button menu to operate on morphs below the surface and can be used to insert or remove morphs from a composite morph. Holding the left mouse button over the squares along the left side of the core sampler highlights the associated morph. As a shortcut, holding down the shift key while pressing the left mouse button over one of these squares will extract the associated submorph (and all its submorphs) from the composite morph. Holding the shift key while dropping a morph onto one of these squares will insert the morph as a submorph of the associated morph. The iconic buttons on the right show the current resizing attributes of the associated morph, and allow them to be changed. (Resizing attributes are discussed in section 7.7.5.)

7.3 Morph Traits and Prototypes

Morphic organizes morphs into a hierarchy much like a class hierarchy. The behavior for all “instances” of a given morph “class” is defined in a shared traits object which is a parent of all the instances. The structure of an instance is defined by the slots of its prototype. The root of the morph hierarchy is `traits morph`. All morphs inherit from `traits morph`, either directly or via one or more intermediate traits objects.

Note: To see the entire morph hierarchy, invoke the “*Show Traits Family*” menu command on an outliner on `traits morph` in the user interface. Be patient; the morph hierarchy is quite large.

Usually, the prototype of a given morph contains all the slots of the morph from which it is derived (the “instance variable of its superclass” in a class-based system) plus, possibly, a few additions. To simplify the life of the programmer, the programming environment supports an idiom known as “copying down.” The derived prototype is described differentially. That is, it is “just like its copy-down parent except with particular slots added, deleted, or changed.” Typical class-based languages also describe the structure of subclasses differentially: a subclass may extend its superclass with additional instance variables. However, the copy-down idiom also allows the derived prototype to selectively omit slots of its copy-down parent or change their contents. For example, the `circleMorph` prototype is derived

from the morph prototype, but it replaces the “rawBox” slot of the morph prototype with the slots “radius” and “center”. This would not be possible in most class-based languages.

Note that the class-like organization of the morph hierarchy — with its parallel traits and prototype hierarchies and its use of the copy-down mechanism to propagate slot information down the prototype hierarchy — is only one way that Self programs can be organized. Other parts of the system, such as the world-wide-web browser, are organized differently.

7.3.1 Implementing a New Kind of Morph

It is easy to make a new kind of morph. One typically starts with a copy of some existing morph and adds or overrides state and behavior to create the new morph. Often, the most appropriate starting point is a copy of *morph*, the root of the morph object hierarchy. Morph has default behavior for everything from drawing to handling user inputs; one thus starts with a working morph and modifies its behavior incrementally to create the new type of morph.

The programming aspect of creating a new morph is straightforward. However, four other things must be done to make the new morph into a first-class citizen. First, its behavior should be factored into a shared parent (called a traits object) to allow the behavior of all instances to be changed by changing the shared parent. Second, the shared traits object and a prototypical instance of the new morph should be embedded in the global namespace. Third, the copy-down parent of the prototype should be set so that changes to the structure of the parent are propagated correctly. Finally, the new prototype and traits objects should be assigned to a module to allow the code for the new morph to be saved in a file.

Of course, if one just wants to do a quick experiment, none of these housekeeping chores are necessary. However, sometimes one decides to make an experimental morph into a first-class morph (the bottom-up approach). In other cases, one sets out from the beginning to create a new first-class morph (the top-down approach). The next two sections will describe how to create a new kind of first-class morph using each of these approaches.

7.3.2 Morph Creation: The Bottom-up Approach

In the bottom-up approach, one is initially interested in getting a morph with the desired behavior as quickly as possible. Thus, an appropriate morph is copied and modified by adding slots directly to the morph itself. Suppose one wished to create a morph that displayed as an oval and that toggled between two colors when the middle mouse button was pressed. To get a morph to modify, evaluate:

```
morph copy
```

This will make an outliner on a new morph. Use the “*Show Morph*” command on this outliner’s middle-button menu to make the graphic representation of the copy appear.

The “*Add Slot*” command on the outliner’s middle-button menu can be used to add a data slot to hold the alternate color. Enter the following expression and accept it by clicking on the green (top) button:

```
otherColor <- paint named: 'leaf'
```

The morph’s drawing behavior can be customized by adding the method:

```
baseDrawOn: aCanvas = (  
  aCanvas fillArcWithin: baseBounds  
    From: 0  
    Spanning: 360  
    Color: color.  
  self)
```

Morphic optimizes shadow drawing for rectangular morphs such as prototypical morph, which draws as simple rectangle. However, this morph is not rectangular. To make its shadow reflect its true shape, the *isRectangular* behavior must be overridden by adding the constant slot:

```
isRectangular = false
```



```

◀ an object<8>(type: morph)
  Modules: morph, -, morphSaving
  parent*                               traits morph
  baseDrawOn: aCanvas                    ...
    aCanvas fillArcWithin; baseBounds
      From: 0
      Spanning: 360
      Color: color.
  self
  isRectangular                          false
  middleMouseDown: evt                    ...
    | tmp |
    tmp: color.
    color: otherColor.
    otherColor: tmp.
    self
  otherColor                             a paint<357>(leaf)
  ▶ Basic Morph State
  ▶ filing out

```

Fig. 7.4: A new kind of morph has been created by modifying a copy of the standard morph. The slots `baseDrawOn:`, `isRectangular`, `middleMouseDown:`, and `otherColor` have been added to the morph to obtain the new behavior. The morph itself appears on the left; an outliner showing its slots appears on the right.

The new morph's input behavior can be customized by adding the method:

```

middleMouseDown: evt = (| tmp |
  tmp: color.
  color: otherColor.
  otherColor: tmp).

```

The morph now draws itself as a filled oval. Clicking the middle mouse button on it causes its color to toggle between its original color and leaf-green. This new morph can be used as a prototype; any copies will get the state and behavior of the prototype at the time of copying. However, later changes to the prototype will not be reflected in the copies. For example, even if the prototype's middle-mouse behavior were changed to cycle through three colors, copies made before this change would still only toggle between two colors.

To allow the behavior of all copies to be changed at once, one can move shared behavior and state into a shared traits object. Self's object literal syntax can be used to create a new object to be used as the shared traits. As a shortcut, rather than creating an empty traits object and then adding a parent slot to it, an object containing an initialized parent slot can be created in a single operation. Evaluate the expression in an evaluator on the outliner (use the middle-button

menu command “*Evaluator*” to open an evaluator on the outliner if necessary):

```
(| parent* = traits morph |)
```

This creates a new object whose parent is traits morph. To make the new morph inherit through the new traits object, invoke the “*Grab pointer*” command by pressing the middle-button menu on the button on the right side of the parent slot of the morph. Drop the end of the pointer onto the new traits object to make the parent slot of the morph point to the new morph. This technique is called “arrow-dragging”.



Fig. 7.5: A new traits object has been created to hold behavior to be shared by all instances of the morph. Arrow dragging is being used to make the parent slot of the new morph point to the new traits object.

Now, the behavior to be shared can be moved from the prototype into the new traits object. Invoke the “*Move*” command on the middle-button menu for the `isRectangular` slot. This causes the slot to be plucked out of the object. Drop the slot onto the shared traits object. This causes it to be added to that object. Repeat this process for the `baseDrawOn:` and `middleMouseDown:` slots.

The Self language uses slot inheritance to share both data (`isRectangular`) and behavior (`baseDrawOn:` and `middleMouseDown:`). The programming environment supports a similar kind of uniformity by allowing any slot to be moved or copied by via drag-and-drop. A entire category can also be copied or moved by dragging.

Note that when moving a slot between an object and its parent there is an interval during which the slot is not in either object. If a message matching the slot name is sent to the object during this interval, the object's response is be determined by a slot inherited from a parent higher in the inheritance chain, if any. If it is important to avoid this transient state, one can move the each slot by first copying it from the prototype into the parent and then remove it from the prototype. A slot is removed simply by moving it and dropping it onto the trash can morph (or by dropping it on the background and then dismissing it).

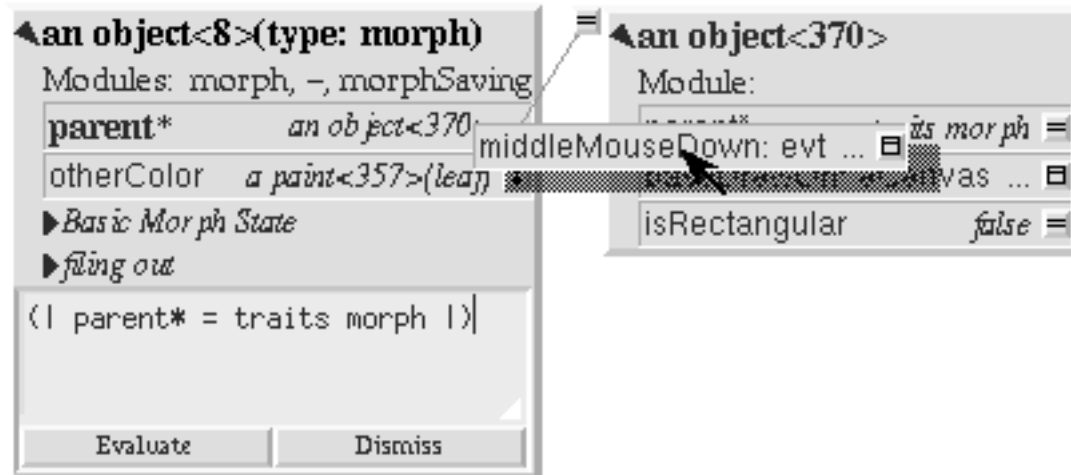


Fig. 7.6: Using slot-dragging to move a slot into the new traits object. Using direct manipulation to move and copy slots makes programming feel like manipulating concrete objects. This narrows the gap between composition of graphical objects (building and modifying composite morphs) and programming.

Now, changing the traits object changes the behavior of all instances. For example, the draw method in the traits can be changed to draw an unfilled oval. To demonstrate the power of shared behavior, first make several copies of the prototypical oval using the “Duplicate” command on its right-mouse menu. Then modify the `baseDrawOn:` method in the shared traits as follows (note the change from `fillArcWithin:` to just `arcWithin:`):

```
baseDrawOn: aCanvas = (
  aCanvas arcWithin: (baseBounds indent: 3)
    From: 0
    Spanning: 360
    Width: 3
    Color: color.
  self)
```

The oval is drawn with a pen three-pixels wide. To accommodate the extra width, the rectangle passed to the canvas is indented by three pixels. Note: A morph should never draw outside its `baseBounds`. When this method is accepted, all copies of the prototype reflect the change. However, Morphic doesn't automatically redraw instances when the draw method is changed. To see the change, drag some large object over the ovals to make them redraw.

At this point, the prototype for a new kind of morph has been created and the behavior common to all its instances has been factored into a separate traits object to facilitate later changes. The next step is to install the prototype and its traits in the global and traits namespace objects. Doing this allows the morph prototype and its traits to be referred to by name.

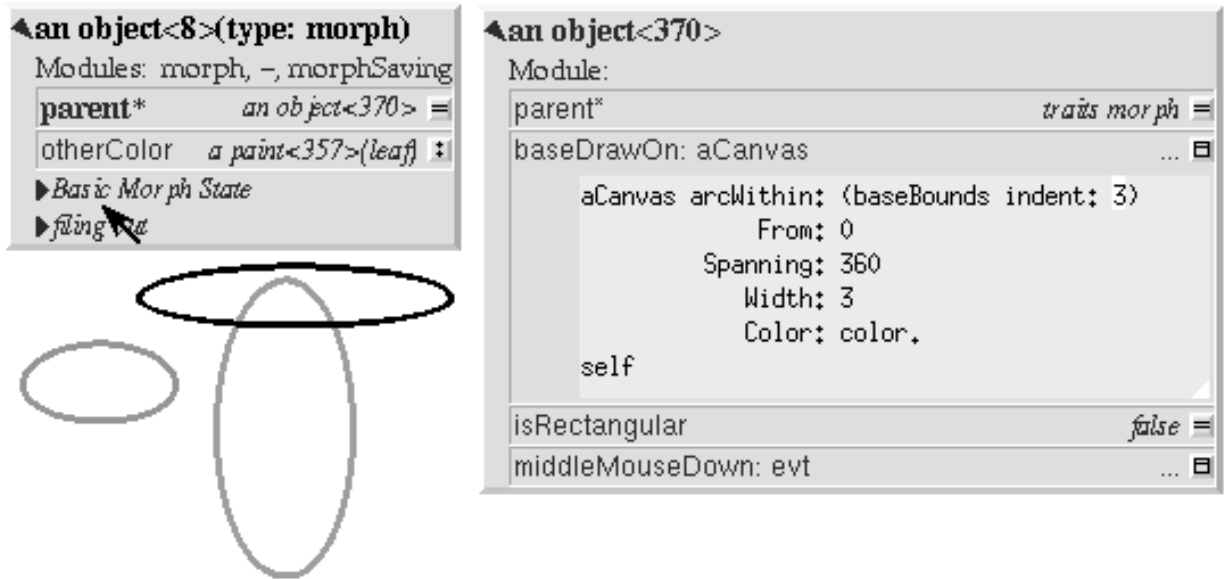


Fig. 7.7: Changing the shared traits object changes the behavior of copies of the prototype (instances). In this case, the draw method has been changed to draw unfilled ovals.

First, summon outliners for the globals and traits namespace objects by evaluating the expressions `globals` and `traits` in any text editor. (That is, type the expression, select it, and invoke the “*Get Expression*” command in the middle-button menu of the editor.) Then, open an appropriate category for the new morph or create a new category. Within the chosen category of `globals`, create a slot to hold the new morph’s prototype by invoking the “*Add Slot*” command and accepting the following expression:

```
ovalMorph = nil
```

Next, invoke the “*Grab pointer*” command by pressing the middle-button menu on the button on the right side of the new `ovalMorph` slot. Drop the end of the pointer over the new morph prototype and release the mouse. This makes the new slot point to the new morph prototype. Repeat the procedure just described to create an `ovalMorph` slot in the traits namespace and point it to the traits object for the new morph.

Finally, invoke the “*Make creator*” middle-button menu command on each new `ovalMorph` slot to designate it as that morph’s creator. This informs the system that the given slot is the given object’s home in the global namespace. (An object may be reachable via several global slots; setting its creator path distinguishes one of these slots as the object’s official “home address.” This information is used to determine the object’s name, as well as which the module in which to record information about the object as a whole, such as the object comment.) In a few seconds (if outliner updating is on), the outliner titles of the `ovalMorph` prototype and its traits object will be updated to show the new names for these objects.

To allow a composite morph containing `ovalMorhs` to be saved in a file, the prototype method in the prototype (not the traits!) must return the prototype `ovalMorph`. The `ovalMorph` prototype already has a prototype method that was copied from the original morph prototype. Change the body of the prototype method in the “*filing out*” category to:

```
ovalMorph
```

Many Smalltalk programming environments allow an instance variable to be added to a class at runtime. The new instance variable is propagated down to all subclasses and added to all existing instances of the class and its subclasses with an initial value of `nil`. The Morphic programming environment can provide a similar service for the copied-down slots of prototypes, with two significant differences: (1) changes to the values of a copied down slots are propagated, as

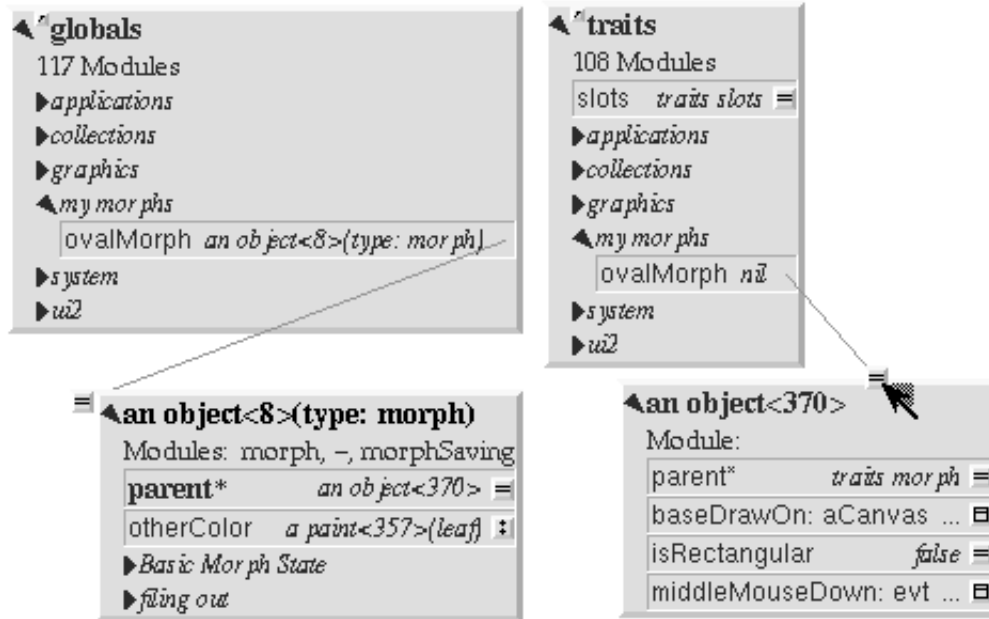


Fig. 7.8: Installing the new morph prototype and traits object in the globals and traits namespace objects. In each case, a new constant slot is created in the appropriate category, then arrow-dragging is used to make the new slot point to the desired object.

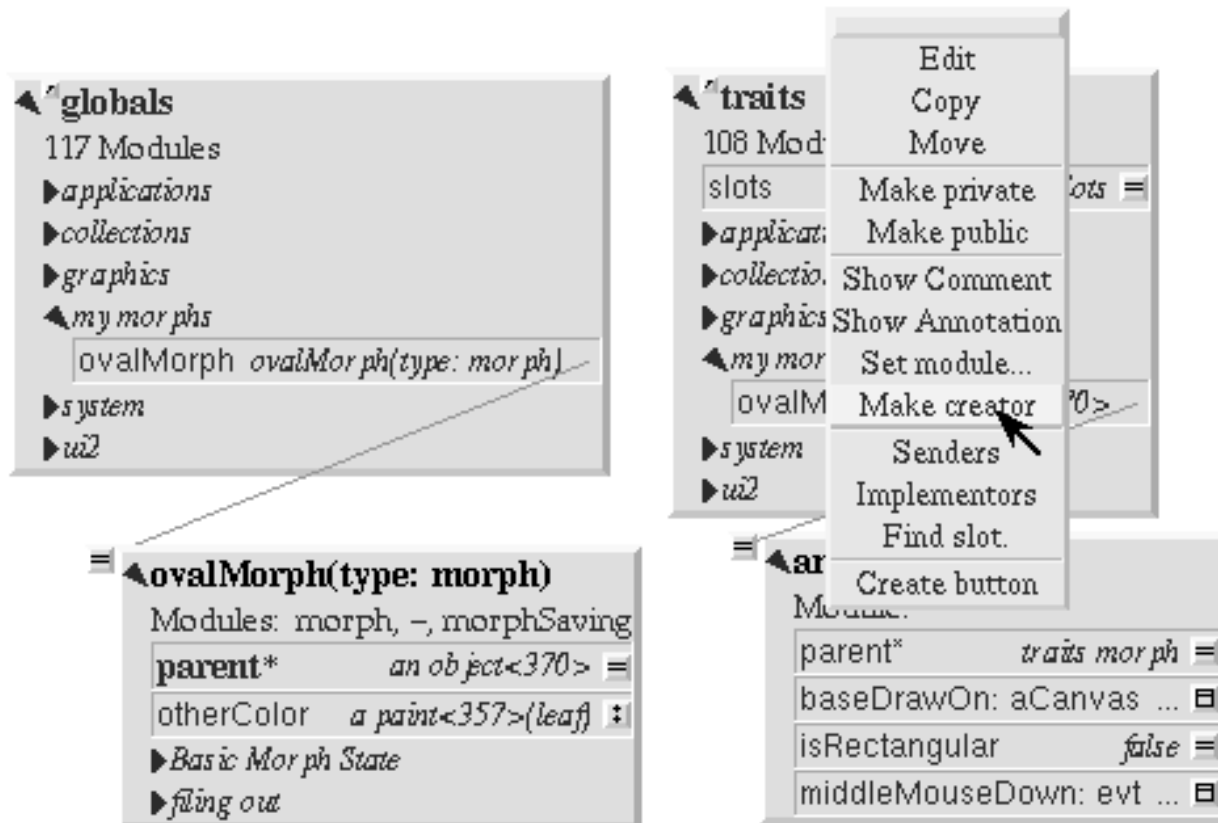


Fig. 7.9: Setting the creator slot of the new traits object. The system uses this information to name objects, among other things. Note that the title of the prototype (on the left) has already been updated to reflect its new name.

well as slot additions and removals and (2) changes are propagated only to objects registered in the global namespace (other prototypes), not to clones of those objects (instances).

The system can be told to maintain the copied-down slots of the `ovalMorph` prototype automatically by setting its copy-down parent (Fig. 7.10). Select the “*Show Annotation*” command in the middle-button menu on the title of the `ovalMorph`’s outliner. Set the copy-down parent field to “`morph`”, the copy-down selector to `copyRemoveAllMorphs`, and press the green (top) button to accept this change. (The copy-down selector is sent to the copy-down parent to create a fresh copy from which to copy slots.) The system will ask if the slots “parent” and “prototype” should be omitted from the copy-down operation, since their contents differ from that of that of the copy-down parent. They should be.

Finally, it would be nice to be able to save the prototype and traits for the new `ovalMorph` in a file so that it can be archived or read into another Self world. Several steps are required. First, the module itself must be created. The system will create a new module (after getting confirmation from the user) the first time its name is used. Then, the slots in the globals and traits namespace object must be assigned to the new module. Finally, the non-copied-down slots in the prototype and traits objects are assigned to the module. This may sound tedious, but the system provide several shortcuts to accelerate the process.

To set the module for the new morphs home slot, invoke “*Show annotation*” on the `ovalMorph` slot in the globals object, type “`ovalMorph`” in the module field, and accept the change (Fig. 7.11). Since this is a new module, the system will ask whether a new module should be created (yes), whether it should be a submodule of an existing module (no), and what subdirectory it should be stored into (“applications”). Set the module of the `ovalMorph` slot in the traits object the same way.

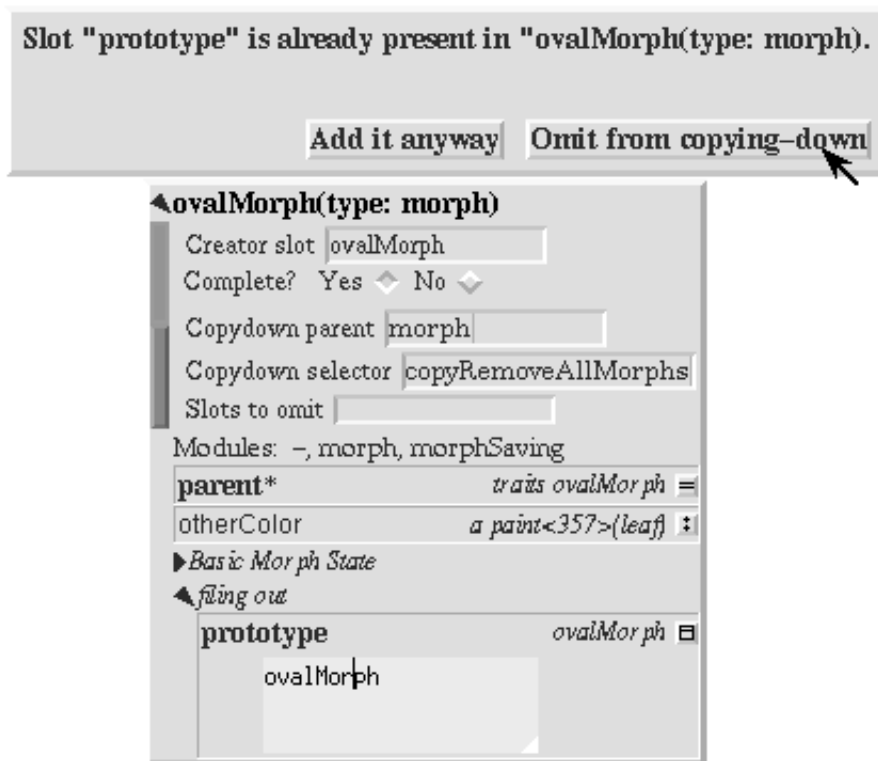


Fig. 7.10: Setting the copydown parent for the new prototype.

All the slots in an object (or within one category of that object) can be assigned to a module in a single operation. To assign the slots of the new traits object to the new module, invoke the “*Set module...*” command on the header of its outliner (Fig. 7.12). The system will ask which slots should be assigned to the module (all) and which module to

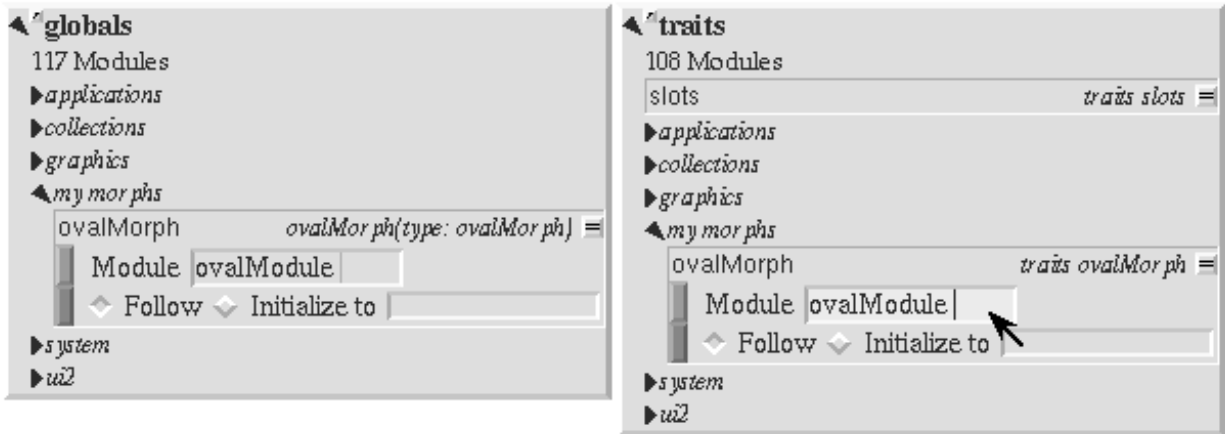


Fig. 7.11: Setting the modules for the namespace slots. The module will be created if it doesn't already exist; the system asks the user several questions about where the new module should live and whether it is a submodule of some existing module.

put them into (`ovalModule`). After a few seconds, the module summary at the top of the outliner should update to indicate that all slots of the traits object are in `ovalModule`. Repeat this procedure to assign all the slots of the `ovalMorph` prototype to `ovalModule`.

Now that all the slots of the new morph and its prototype have been assigned to the new module, the module can be filed out. Invoke the “*Changed modules...*” command on the background menu to get the changed modules morph. Then press the little button marked “w” (Fig. 7.13) to the right of `ovalModule`. The system will save the code for the module in a file named “`ovalModule.self`” in the “`applications/`” subdirectory of the current working directory. (If this directory doesn't exist, the system will complain. Create the directory and try the fileout operation again.) The oval morphs module can later be loaded into a snapshot by evaluating the expression:

```
bootstrap read: 'ovalModule' From: 'applications'
```

7.3.3 Morph Creation: The Top-down Approach

The top down approach to creating a new morph is similar to the approach just described, except that one plans to make a first-class citizen from the beginning. Thus, the order of steps is slightly different. Here is a brief outline of the procedure:

1. Add a slot to the traits namespace object (using “*Add Slot*”):

```
ovalMorph = (| parent* = traits morph |)
```

2. Add a slot to the globals namespace object:

```
ovalMorph = (| parent* = traits ovalMorph |)
```

3. Make each new slot be the creator of its contents (using “*Make creator*”).
4. Set the copy-down parent of the prototype to morph (via “*Show annotation*” on its outliner).
5. Set the module of the two namespace slots to `ovalModule` (creating the new module in the process).
6. Assign all slots of the new traits and prototype objects to `ovalModule`.
7. Start programming the new behavior.



Fig. 7.12: Assigning all the slots of the new traits object to the new module.

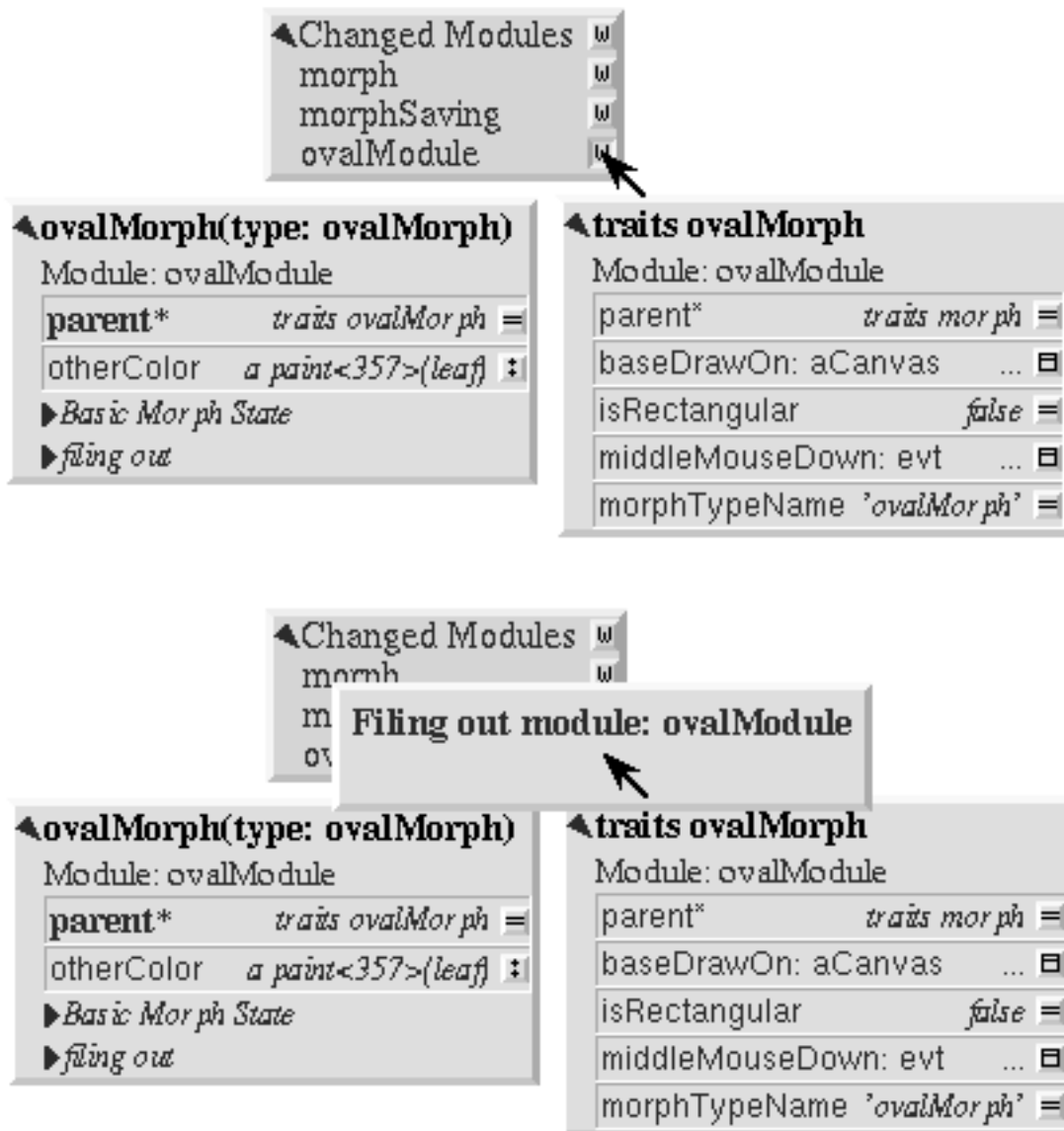


Fig. 7.13: Saving the code for the new morph in a module file.

This procedure does all the housekeeping chores up front, so the module can be filed out at any time. As the programmer works, the system will deduce that any slots added to `ovalMorph`'s traits or prototype should be placed in the same module as the other slots in that object (`ovalModule`). In a future release of the system, the initial housekeeping may be automated. This would make creating a new kind of morph a one-step operation.

7.4 Saving a Composite Morph

The system includes an experimental facility for store the structure of a composite morph to a file. This allows a morph constructed by direct manipulation to be saved into a file that can be read later to reconstruct the morph. This is how the “factory” was created. The morph saving facility requires that every morph and submorph to be saved supply implementations of the messages `slotsToNotFileOut`, `appendOtherSlotsOnto:`, `storeStringNeeds`, and `prototype`. Unfortunately, because morph saving was added later as an experiment, not all morphs have been retro-fitted with implementations of these messages. The enterprising user could easily infer how to add the required support to morphs that do not yet have it.

Suppose one has created a column of useful buttons that one wishes to save. (Fortunately, buttons, columns, rows, frames, and labels are among the morphs that do support saving.) To save this morph, create an outliner for it and then evaluate in that outliner:

```
saveMorphInFile
```

The system will prompt for a file name and will give graphical feedback as each component morph is stored. The file can later be read by evaluating:

```
worldMorph loadMorphFromFile
```

Again, the system will prompt for the file name. A copy of the morph that was saved will be added to the hand. The return value of the expression will also be added to the hand, which may temporarily hide the new morph. Click any mouse button to put down the two objects, then move the top one out of the way.

7.5 Handling User Input

7.5.1 Handling Events

Morphic represents user actions such as pressing a key or mouse button using `ui2Event` objects. A `ui2Event` actually carries two kinds of information: its *type*, such as `leftMouseDown`, and the state of the mouse buttons and certain keyboard keys when the event occurred. This allows a program to tell, for example, if the shift key was held down when the left mouse button was pressed. As events occur, they are placed into a buffer. Morphic removes and processes events from this buffer in order. Thus, even if a user occasionally gets ahead of the system, the system will eventually catch up.

A morph can handle a given kind of event simply by implementing one of the following messages:

```
keyDown: evt
keyUp: evt
mouseMove: evt
leftMouseDown: evt
leftDoubleClick: evt
leftMouseUp: evt
middleMouseDown: evt
middleDoubleClick: evt
middleMouseUp: evt
rightMouseDown: evt
```

```
rightDoubleClick: evt
rightMouseDown: evt
```

The event is always supplied so that its state can be examined. The default behavior of the `leftMouseDown:` message is to pick up the composite morph containing the morph that gets the event. (That is, the left mouse button generally means “move”.) The default behavior of the `rightMouseDown:` message is to pop up the morph menu (the “blue” menu). The default behavior of the other messages is to return the special `dropThroughMarker` object, indicating that the event is not processed by this morph.

Submorphs of a morph are displayed in front of their owning morph. By default, submorphs are usually given the first opportunity to handle incoming events. If a submorph does not handle an event, it returns the `dropThroughMarker` object, and Morphic gives the submorph behind it a chance to handle the event. Each user generates events at the current location of their cursor. One can visualize an event as “falling down through” the submorphs of the composite morph at that location until either the event lands on a submorph that handles it or until all the submorphs of the composite at that point are exhausted. However, events do not fall between top-level morphs. For example, if an outliner is covered by a morph that does not handle `middleMouseDown` events, one cannot invoke the middle button menu of the outliner through the intervening morph.

In some cases, a morph may wish to handle certain events before its submorphs. For example, a `ui2Menu` morph handles `leftMouseDown` events itself rather than letting its component buttons get them in order to highlight the button under the cursor and to pop down the menu when a selection is made. A morph can arrange to handle certain kinds of events before its submorphs by overriding the `allowSubmorphsToGetEvent:` message.

There are actually two classes of events. `keyDown` events, the three `mouseDown` events, and the three `doubleClick` events are dispatched using the “falling through the submorphs” technique just described. The other events — `keyUp`, `mouseMove`, and the three `mouseUp` events are dispatched only to interested *subscribers*. The rationale is that some morphs are interested in discrete events, such as `mouseDown` transitions, while others need to track the mouse or keyboard over an extended period of time. Dispatching high-frequency events such as `mouseMove` to uninterested morphs would be inefficient. Furthermore, some morphs need to get events even when the cursor is no longer over the morph. For example, a click-to-type editor should continue to get `keyDown` events until another editor is clicked. In short, Morphic supports both spatial and subscription-based event dispatching because both are useful.

The events generated by a particular user are dispatched from the `handMorph` associated with that user. Each `handMorph` keeps a list of subscribers interested in various kinds of events. A morph asks the appropriate `handMorph` to start or stop its subscription to a particular kind of event. Every event has a reference to the hand that generated that event. Thus, a morph that wishes to track the mouse until the button is released (e.g., `sliderMorph`) would do the following:

1. on `leftMouseDown`, execute `evt sourceHand subscribeUntilAllUp: self`
2. on `mouseMove`, update the slider position from the current mouse position (which is in global coordinates)

7.5.2 Mapping special characters to actions

When a morph receives the `keyDown:` message, the next step is the interpretation of any control-, meta- or command- keystrokes. For example, on the Macintosh, a command-X should perform a cut operation. A morph wishing to respect these conventions should do two things: it should inherit from `traits ui2Event ignoreSpecialCharactersMixin`, and it should, upon receiving the `keyDown:` message, send `sendMessageToHandleKeyboardEventTo:` the event, passing itself as the argument. The latter message tells the event to decode any special characters and send an appropriate message back to its argument. The mixin provides default behavior.

7.6 Drag and Drop

A morph can perform some action when another morph is dropped onto it and can decide which dropped morphs it will accept. In addition, the dropped morph can perform some action in response to being dropped.

To accept dropped morphs, a morph must respond affirmatively to the message:

```
wantsMorph: m Event: evt
```

The morph to be dropped is supplied as an argument to allow the receiving morph to decide if it wishes to accept the drop. For example, a printer icon morph might accept only document morphs. If the target morph agrees to accept the dropped morph, the target is then sent the message:

```
addDroppingMorph: m Event: evt
```

to actually perform the drop action. Part of this action should be to put the dropping morph somewhere or delete it. For example, the printer icon morph might queue a print request, then add the document morph to a folder morph representing the printed documents.

Finally, the dropped morph is informed of the drop (post facto) by sending it the message:

```
justDroppedInto: m Event: evt
```

The event is provided in these messages to allow the morph to examine the state of the mouse buttons or modifier keys at the time of the drop.

7.7 Automatic Layout

7.7.1 Packing

Automatic layout relieves the programmer from much of the burden of laying out the components of a composite morph such as a dialog box. By allowing morphic to handle the details of placing and resizing the components, the programmer can focus on the *topology* of the layout, without worrying about the exact positions and sizes. Automatic layout also allows composite morphs to adapt gracefully to size changes, including font size changes.

Layout morphs manage the placement and sizing of their submorphs. Layout morphs currently include `rowMorphs`, `columnMorphs`, `frameMorphs` and their descendents. All other morphs leave the size and placement of their submorphs alone. The current set of layout morphs all use the same layout strategy: linear, non-overlapping packing. Rows pack horizontally from left-to-right. Columns, frames, and their descendents pack vertically from top-to-bottom. This simple approach, while it does not handle every conceivable layout problem (e.g., tables whose rows and columns adjust to the size of their contents), is surprisingly powerful. All automatic layout in morphic is based on nested combinations of rows and columns.

Linear packing is best explained procedurally. Consider a `rowMorph`. Its task is to arrange its submorphs into a row such that the left edge of each morph just touches the right edge of the next morph. The submorphs are processed in order; that is, the first submorph will be placed at the left end of the row, then the next submorph will be placed to the right of the first, and so on. The last submorph will be placed at right-most end of the row. Notice that the order of the submorphs is not affected by the packing process. Also notice that the packing is done only in one primary dimension—the horizontal dimension in this case. The other dimension is also considered during packing, and is controlled by the *justification* parameter of the row. Depending on this parameter, the tops, bottoms, or centers of the submorphs can be aligned with the top, bottom, or center of the row.

7.7.2 Space-filling

For simplicity, the packing strategy was described as if the submorphs to be packed were all rigid. In order to support “stretchy” layouts, morphs can be designated as *space-filling*. (Note: The source code uses the older term, *flexible*.) When there is extra space, a space-filling morph expands to fill this space. If there is no extra space, a space-filling morph shrinks to its minimum size. When there are several space-filling morphs in a single row or column, any extra space is divided evenly among them.

Space-filling morphs can be used to control the placement of submorphs within the primary dimension when a row or column is stretched. For example, suppose one wanted a row with three buttons, one at the left end, one at the right end, and one in the middle. This can be accomplished by inserting space-filling morphs between the buttons:

```
<button1><spacer><button2><spacer><button3>
```

When the row is stretched, the extra space is divided evenly between the two spacers, *button2* stays in the center, and *button3* stays at the far right. By making the color of the spacers match that of the underlying row, they become effectively invisible. This is a common technique.

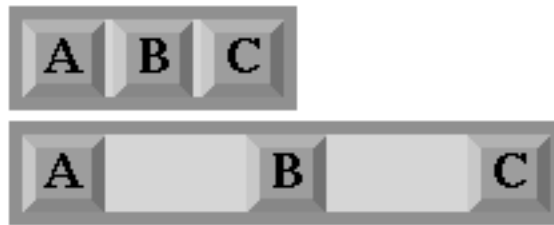


Fig. 7.14: Using flexible spacer morphs to space buttons evenly within a row. Normally these spacers would be made the same color as the row, making them effectively invisible

7.7.3 Shrink-Wrapping

It is sometimes desirable for the size of a morph to depend on the sizes of its submorphs. For example, the size of a button should depend on the size of its label. (It would be annoying if it didn’t; the programmer would have to manually resize the button after changing the label.) A morph designated as *shrink-wrap* shrinks (or grows) to the smallest size that accommodates the size requirements of its submorphs.

7.7.4 Minimum Sizes

Morphs have a minimum size in each dimension (`minWidth` and `minHeight`). These sizes determine the minimum amount of space that will be allocated to a morph during layout. The minimum size of a morph takes into account the minimum sizes of its submorphs. For example, the minimum width of a row is the sum of the minimum widths of its submorphs (plus a little bit for a border, if it has one).

The absolute minimum width and height of a morph, even when it has no submorphs, is specified by its `baseMinWidth` and `baseMinHeight`. For some kinds of morph, these values are stored in assignable slots in the morph. For others, these values are defined by inherited constant slots to save space. One can use these attributes to give a space-filling morph a minimum size.

7.7.5 Resize Attribute Summary

The resizing behavior of a morph in one dimension is completely independent of its behavior in the other dimension; that is, a morph actually has two independent resizing attributes, one for the horizontal dimension and one for the vertical dimension.

To summarize, the resizing behavior of a morph along a given dimension is controlled by its resizing attribute, which has one of three values:

rigid The morph is not resized.

space-filling In a row or column, the size of the morph adapts to fill the available space. Extra space is shared evenly with any other space-filling morphs in that row or column.

shrink-wrap The morph is shrunk to just fit around its submorphs, or to its minimum size, whichever is smaller. Enclosed space-filling morphs are shrunk if necessary.

A morph's minimum size in a given dimension determines the smallest amount of space that should be allocated to it during layout. The core sampler and/or properties sheet can be used to change these attributes.

7.8 Animation

Animation can be used to make an interactive application seem more alive and can convey valuable information. However, animation can become annoying if the user has to wait until the animation completes before doing anything else. In Morphic, animation and user actions are concurrent, and multiple animations can be active while multiple users interact with the system.

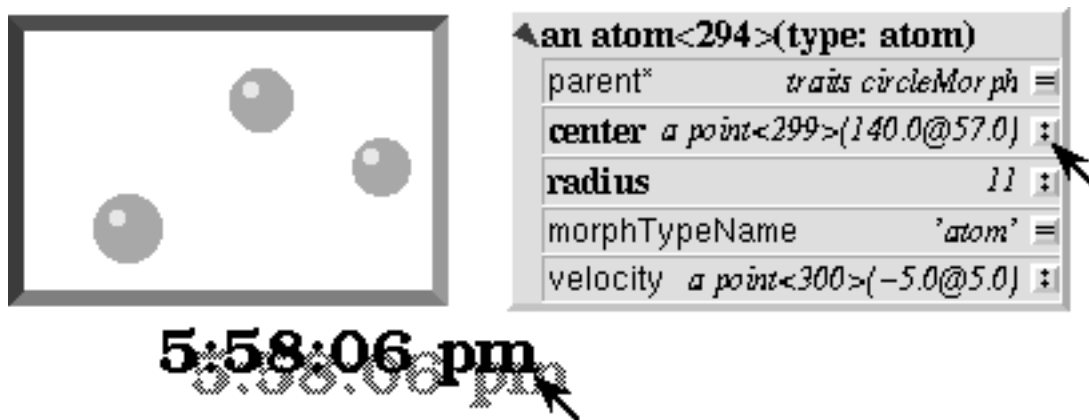


Fig. 7.15: Three simultaneously active morphs: an ideal gas simulation, a digital clock, and an outliner on the Self object underlying one of the atoms in the simulation. The clock updates every second, the simulation runs continuously, and the outliner periodically updates its center and velocity slot values as the underlying atom moves. A morph continues to operate while it is being moved (the clock is being moved here) or while an external animation is applied to it. Note that multiple users can be active simultaneously; this example shows the cursors of two users.

There are two ways to achieve animation. First, a morph can have lightweight autonomous behavior which typically, although not necessarily, appears as animation. For example, a clock might advance the time or a discrete simulation might compute simulation steps. Second, Morphic includes a kit of external animation behaviors that can be applied to any morph, including motion, scaling, and color change animations.

Although autonomous behavior and external animations are implemented using the same underlying mechanism, they have different purposes and are specified in different ways. The autonomous behavior of a morph is an intrinsic property of that morph. For example, updating the time is central to being a clock morph. Autonomous behavior is

defined in the morph itself. External animation behaviors, on the other hand, are typically transient and imposed from outside. For example, the Self programming environment gives feedback for certain actions by “wiggling” the relevant morph. An external animation is specified by creating a separate animation activity object and applying it to the morph to be animated. Animation is orthogonal to autonomous behavior; for example, a clock morph would continue to run even while a motion animation whisked it across the screen.

7.8.1 Stepping

The autonomous behavior of a morph is defined by its `step` method. For example, to make a simple digital clock, one could add the following slot to a copy of `labelMorph`:

```
step = ( label: time current timeString )
```

The clock is activated by asking the system to send the “`step`” message to it either continuously (every display update cycle) or at periodic intervals (e.g., once per second). Make sure the `labelMorph` is visible in the world (use the “*Show Morph*” menu command if necessary), then, in an evaluator on its outliner, evaluate:

```
getSteppedEveryMsecs: 1000
```

This will cause the `step` message to be sent to the morph once per second (i.e., every 1000 milliseconds), causing it to display a formatted string representing the current time. `step` messages are sent synchronously during the display update cycle. This has the advantage of simplifying synchronization but requires that `step` methods complete quickly to avoid delaying user interactions.

The message `stopGettingStepped` can be sent to the morph to turn off stepping for that morph. Morphic automatically stops stepping when the target morph is removed from the world. To make the clock morph reactivate itself when dropped back into the world, add the following slot:

```
justDroppedInto: m Event: evt = (
  isInWorld ifTrue: [ getSteppedEveryMsecs: 1000 ] ).
```

7.8.2 External Animation

External animation of a morph is achieved by scheduling an *animation activity* with that morph as its target. An animation activity changes some property of its target gradually over the course of a number of display cycles (frames). For example, a `positionAnimator` animates a change in its target morph’s location. The programmer specifies the initial and final values of the property to be changed (e.g., the starting and ending position) and the duration over which the change should occur. The duration can be defined in two ways. *Frame-based* animation lets the programmer control animation smoothness by specifying that the animation should take a given number of frames regardless of the time per frame. *Time-based* animation lets the programmer specify the desired amount of time the animation should take, but the number of intermediate frames depends on the time per frame, which may vary with system load, scene complexity, and other factors. Animations can be paced linearly or slow-in-slow-out. A slow-in-slow-out animation starts slowly, builds to a maximum pace, then decelerates. There are activities that animate the position, size, and color of morphs, activities that send arbitrary messages, and compound activities that combine a set of other activities either sequentially or concurrently. In fact, this activity architecture is the basis of all animation in Morphic: an activity called a `periodicStepActivity` is used to implement the stepping facility.

7.9 Other Issues

7.9.1 Local versus Global Coordinates

The position of a morph is defined relative to the position of its owner. This makes it unnecessary to update the positions of all the submorphs when moving a composite morph. However, it also means that morphs with different owners have positions in different coordinate systems. In order to compare the positions of morphs having different owners, it is necessary to use their positions in the world’s coordinate system, which are computed by sending the `globalPosition` message to each morph.

7.9.2 Synchronization

Animation, stepping, and other activities are handled synchronously, as part of the basic user interface loop. Thus, a sequence of actions done by an activity or a `step` method appear to happen atomically; the user never sees the morph in an intermediate state in which some but not all of the actions have taken place. For example, if a morph is removed from one morph and added to another, the user never sees the transient state in which the morph is not in the world at all. Likewise, any layout modifications resulting from user actions—such as adding a new morph to a row—appear to happen atomically; one never sees a partially complete layout.

Often, however, an independent Self thread wishes to manipulate morphs in the user interface. In order to make such actions appear atomic, they should be done under the protection of the UI synchronization semaphore. The preferred way to do this is to wrap the action or actions in a block to be executed between display cycles of the morph’s world:

```
aMorph safelyDo: [ ... ]
```

Synchronization errors usually appear as intermittent graphical glitches, although in rare cases the submorph structure may be corrupted (e.g., a morph appearing in the submorph lists of multiple morphs).

7.9.3 Display Updating

Morphic uses a double-buffered, incremental algorithm to keep the screen updated. This algorithm is efficient (it tries to do as little work as possible to update the screen after a change) and high-quality (the user does not see the screen being repainted). It is also mostly automatic; many applications can be built without the programmer ever being aware of how the display is maintained. The description here is mostly for the benefit of those curious about how the system works.

Each morphic screen window displays the contents of some `worldMorph`. A `worldMorph` keeps a list of rectangular “damaged” regions of the screen. Every morph can compute a rectangle that encloses its entire visible representation. When a morph changes its appearance (for example, its color), it sends itself the **message** `changed`. This causes its bounding rectangle to be translated into global coordinates and added to the damage list of the `worldMorph` that contains it. (This `worldMorph` is found by starting at the morph and following the **owner** chain; the `worldMorph` is the last morph in this chain.) On the next display update cycle, the `worldMorph` redraws the portions of all morphs that intersect rectangles in the damage list (via an off-screen buffer), including the morph that was changed. The `worldMorph` then clears its damage list to prepare for future damage reports.

When a morph changes size or position, damage is reported both before and after the change. This causes the screen to be updated at both the old and new size or position.

Typically, the implementor of a morph writes code to send the `changed` message automatically after updating any slot that affects the morph’s appearance. For example, the `color:` message defined in `traits morph` sends `changed` automatically. Likewise, external animation activities report appropriate changes. Thus, the client of a morph usually need not send `changed` explicitly.

7.9.4 Layout Updating

Morphic also maintains morph layout incrementally. When a morph is changed in a way that could influence layout (e.g., when a new submorph is added to it), the message `layoutChanged` is sent to the morph. This triggers a chain of activity. First, the layout of the changed morph is updated. This may change the amount of space apportioned to some of its sub morphs, causing their layouts to be updated. Then, if the space requirements of the changed morph have changed (e.g., if it needs more space to accommodate the newly added submorph), the layout of its owner is updated, and possibly its owner's owner, and so on. In some cases, the layout of every submorph in a deeply-nested composite morph may need to be updated. Fortunately, there are many cases where the layout updates can be localized. Morphic detects these cases, thus saving a great deal of unnecessary work.

As with `changed` messages, morph clients usually need not send `layoutChanged` explicitly since the most common operations that affect the layout of a morph—such as adding and removing submorphs or changing its size—take care of this already.

Normally, layout is performed incrementally after every morph add or remove operation. However, when a large composite morph is to be constructed, the cost of the repeated layout operations can be significant. The programmer can ameliorate this problem by using the batch operation `addAllMorphs:` rather than a sequence of individual `addMorph:` operations.

If a morph is not in a `worldMorph`, however, all layout is deferred. This is done partly to optimize creating large composite morphs (which are often constructed “off-line” and then added to the world) and partly because the exact size of `labelMorphs` depends on font metrics that may vary from one X server to another. Thus, the layout of a morph containing labels would have to be recomputed in the context of a particular world anyway. Occasionally, one needs to know the exact size of a newly created morph (for example, to ensure that a menu does not pop up partially off the edge of the screen). In such cases it may be necessary to temporarily add the morph to the world in some remote location (such as `-1000000 @ -1000000`) to force it to be laid out.

7.9.5 Morph Copying

When a composite morph is copied, its entire submorph tree is traversed and copied to produce a duplicate with the same structure. However, simply copying the structure is not quite enough because some of the morphs within a composite morph may refer to other morphs within the composite. For example, the buttons of a `radarView` refer to the `radarDisplay` morph. When a `radarView` morph is copied, the buttons of the copy must be updated to point to the `radarDisplay` morph in the copy, not that in the original `radarView`. A simplified diagram of this process is shown in [Fig. 7.16](#).

Sometimes a morph may need to do something special when it is copied. In this case, the message `baseCopy` should be overridden rather than `copy`. See `traits ui2Button` for an example of how this is done.

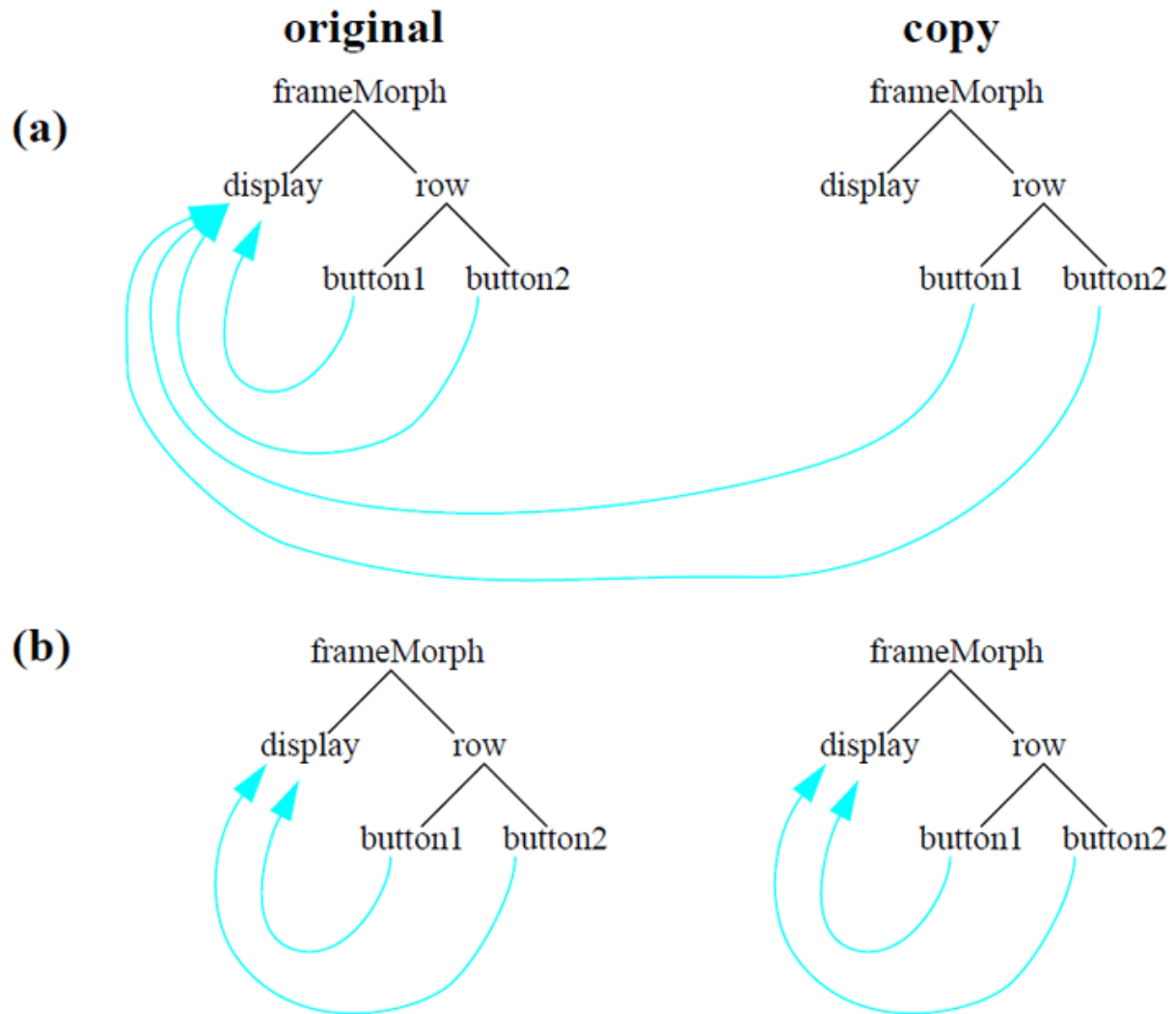


Fig. 7.16: Copying a composite morph. First, the submorph structure of the original morph is copied (a). Then, references among the submorphs of the composite updated to mirror those of the original (b).

7.10 Morph Responsibilities

There are two messages that each type of morph is expected to implement: `morphTypeName` and `prototype`. The first returns a string used to show the type of a morph in the user interface (e.g., in the core sampler) while the second, which should return the prototype for the morph, is used by the morph filing out code.

Two other messages may need to be overridden. These are:

isRectangular This message is used to optimize the drawing of shadows for morphs whose display completely fills their bounding rectangle. The default implementation returns `true`, so non-rectangular morphs such as `circleMorphs` must provide an implementation that returns `false`. (Hint: If a non-rectangular morph casts a rectangular shadow, someone probably forgot to override this message.)

mapReferencesUsing: This message is sent during copying to update any references between the submorphs of a composite morph. Its argument is a dictionary mapping submorphs in the old composite morph to the corresponding submorphs in the copy. Morphs whose slots may contain references to other morphs within a composite should override this message to update these slots during copying. For example, a `ui2ButtonMorph` overrides this message in order to update its “target” slot. That way, if the button and its target are both embedded in some composite morph that is copied, the button in the copy will refer to the target in the copy. See `traits colorChangerMorph` for an example.

7.11 Some Useful Morphs

The Self system comes with a large library of morphs. While some morphs exist solely to supporting the programming environment, many are general-purpose and can be reused to construct new applications. This section mentions some of the most useful and reusable morphs. To find out more about a given morph, use the programming environment to examine its prototype and traits objects. Useful comments are sometimes buried in the bodies of methods.

Widget morphs are interactive, allowing the user to invoke an action or input some data.

sliderMorph Allows the user to specify a numerical value in some range. When the slider is manipulated, its target object is sent a user-specified message with the new slider value as an argument.

ui2Button Executes a user-specified script when the button is pressed. The script can refer to the button’s target. The target of a button or slider morph can be set by using the middle-mouse menu “*Set Target*” command. This sets the target slot of the button or slider to the morph directly below it. Buttons are often decorated with a textual label, but a button can contain arbitrary morphs instead of, or in addition to, this label.

ui2Menu A column full of buttons. A menu can be “pinned down” using the unlabeled button at its top. It can then be manipulated or disassembled like any other morph. Menus support a rich set of messages for adding normal or grayed out buttons and for inserting dividing lines.

Structural morphs are typically used to bind morphs together and arrange them into a pleasing layout.

rowMorph and **columnMorph** Pack their submorphs into a row or column. These morphs offer several justification options and can also provide a border of empty space around their contents.

frameMorph Like a `columnMorph`, except that it can display various kinds of borders around its contents. Bezeled `frameMorphs` are used heavily in the programming environment to provide a three-dimensional look.

spacerMorph While many types of morph (such as an empty `rowMorph`) could be used to fill a space between morphs, it is preferable to use a `spacerMorph` to make it clear that the only purpose of the morph is to control spacing. (Morphic allows users to customize the user interface by directly manipulating morphs. Thus, just as it is important to write readable programs, it is important to build composite morphs with “readable structure.”) Often, a `spacerMorph` is used to provide a fixed amount of space

between submorphs in a `rowMorph` (or `columnMorph`). To accomplish this, the `spacerMorph` should be of the desired width, be rigid horizontally and space-filling vertically, and be the same color as the `rowMorph`. The message **copyH:Color:** (or **copyV:Color:** to creating a vertical spacer for use in a column) can be sent to `spacerMorph` to create a new `spacerMorph` with these properties. The other common use of `spacerMorphs` is to provide a stretchy space between morphs; the expression “`spacerMorph copy beFlexible`” makes a `spacerMorph` that does the job. Setting the **baseMinWidth:** or **baseMinHeight:** of such a spacer ensures that at least the given amount of space will be provided.

Other morphs supply decorative or information content for user interfaces.

labelMorph displays a single-line string in a single font style, size, and color.

circleMorph displays a filled circle.

pixmapMorph displays an image (currently, at most 8 bits deep).

movieMorph cycles through a sequence of images as it is stepped.

The library includes two kinds of text editors.

editorMorph a general editor that allows arbitrary morphs to be embedding in the text.

uglyTextEditor a simple, text-only editor that is a bit faster for editing larger amounts of text.

Many applications implement specialized content morphs. For example, the Self programming environment defines morphs that represent Self objects, slots, and categories.

7.12 The Graphical Environment

Morphic hides many details of the underlying graphics system. This both simplifies programming and provides portability: the layer of abstraction between the programmer and the underlying graphics system allows the implementation of the low-level graphics to be changed without affecting programs written by clients. While the current version of the system is built on the X window system, it could be ported to other window systems fairly easily (although the target window system should support color or grayscale for good results). One might even create a Postscript implementation of the morphic graphics interface to allow morphs to render themselves on paper.

The graphics interface is implemented by canvas objects. There may eventually be many kinds of canvases for rendering onto displays of differing resolutions, color properties, or bit-depths. The current system provides four types of canvas. `windowCanvas` and `pixmapCanvas` draw onto a window or an offscreen buffer via the X protocol. A `nullCanvas` has the same interface but does not actually draw anything; it can be used to factor out the cost of graphics during performance analysis. `colorRecordingCanvas` is used internally by the colormap manager. All canvases implement the following messages for drawing geometric shapes:

Draw a single pixel:

```
point: p Color: c
```

Outline or fill a rectangle or fill the entire canvas:

```
rectangle: r Color: c  
rectangle: r Width: w Color: c  
fillRectangle: r Color: c  
fillColor: c
```

Draw a solid or dashed line or a connected sequence of line segments:


```

line: pt1 To: pt2 Color: c
line: pt1 To: pt2 Width: w Color: c
dashedLine: pt1 To: pt2 DashSize: d Offset: o Color: c
dashedLine: pt1 To: pt2 Width: w DashSize: d Offset: o Color: c
lines: pointList Color: c
lines: pointList Width: w Color: c

```

Outline or fill a polygon:

```

polygon: pointList Color: c
polygon: pointList Width: w Color: c
fillPolygon: pointList Color: c

```

Outline or fill a circle:

```

circleCenteredAt: pt Diameter: d Color: c
circleCenteredAt: pt Diameter: d Width: w Color: c
fillCircleCenteredAt: pt Diameter: d Color: c

```

Outline or fill a wedge cut by the given angles from an ellipse bounded by the given rectangle:

```

arcWithin: r From: startAngle Spanning: spanAngle Color: c
arcWithin: r From: startAngle Spanning: spanAngle Width: w Color: c
fillArcWithin: r From: startAngle Spanning: spanAngle Color: c

```

Draw a simple or compound curve:

```

bezier: pt1 Control: c1 Control: c2 To: pt2 Width: w Color: c
bSpline: controlPoints Width: w Color: c
catmullRomSpline: controlPoints Width: w Color: c

```

Draw text in the given font and size:

```

text: s At: pt Font: fName Size: fSize Color: c

```

Display a portable pixel-based image (a `ui2Image`):

```

image: i At: pt

```

Canvases maintain an offset, allowing graphic operations to be automatically translated. (Canvases also maintain a scale factor, but scaling is not currently used and is probably buggy. Furthermore, image scaling is not implemented.)

In `morphic`, unlike many graphics packages, the graphics context is hidden from the programmer; all the common parameters that control the behavior of a given drawing command—such as color and line width—are passed as explicit parameters. A few infrequently changed parameters, such as the fill pattern and the clipping rectangle, can be changed temporarily via messages such as `withPattern:Do:` and `withClip:Do:.` The canvas handles these messages by changing the state of the underlying graphics context, executing the block provided (which presumably issues some drawing commands to that canvas), and restoring the original state of the graphics context. Nested invocations of `withClip:Do:` are handled sensibly: a stack of clipping rectangles is maintained and drawing operations are clipped to the intersection of all rectangles currently on the stack.

7.12.1 Specifying Colors

Colors in `morphic` are represented by *paint* objects. A paint can be manipulated as either a red-green-blue triplet or as a hue-saturation-brightness triplet. Red, green, blue, saturation, and brightness are specified as numbers in the range [0.0 .. 1.0], where zero means black or unsaturated and one means full-brightness or saturated. Hue, which

corresponds to the angular location of the hue on the color-wheel, is specified as a number in the range [0.0 .. 360.0], where zero corresponds to red. Colors with zero saturation (i.e., black, white, and shades of gray) have no hue; if you increase the saturation of such an achromatic color, its hue is arbitrarily chosen to be zero (red).

Paints provide transformations to:

- change the red, green, or blue component, change the hue, saturation, or brightness component, and interpolate between two colors.

Since paint objects are immutable, all these transformations are *functional*. That is, they return a new paint object, leaving the original paint object unchanged.

Paint objects describe colors in a device-independent and persistent manner. They can be saved in snapshots and filed out, and used with any kind of display (or printer, if printing were supported). The details of color map management are handled by each kind of canvas in a way appropriate for the underlying medium. For example, a canvas for a gray-scale display might map colors to shades of gray according to brightness.

7.12.2 Specifying Fonts

When drawing text in morphic, the font's name and size are specified independently. The size parameter specifies the font height in pixels, and typically ranges from 6 to 72 or more. (The capital letters of a 72 pixel font are about an inch high on a typical display.) This interface suggests that the underlying graphics system fonts can be scaled arbitrarily and, indeed, many modern X servers do support scalable fonts.

The scheme that was implemented for Self 4.0 has not survived the Macintosh port¹. In order to allow portable specification of fonts, we have introduced a `fontSpec` prototype that holds a font's family name (e.g. `times`), a font style (e.g. `bold`), and a font size (e.g. `12`). This object uses an immutable public protocol; it responds to `copy-Name:`, `copyName:Style:`, `copyName:Style:Size`, etc. Once you have created a `fontSpec` object you can then pass it to, for example, a label morph:

```
myLabel fontSpec:  
  fontSpec copyName: 'helvetica' Style: 'bold' Size: 14
```

`fontSpec`'s encapsulate some attributes of a font and in the future should perhaps encapsulate the color as well.

¹ The rest of this section has been written in 1999 under time pressure to get Self 4.1 out so I can get back to other things. John bears no responsibility for its shortcomings. You can send questions about this to me, David Ungar, at david.ungar@sun.com.

VIRTUAL MACHINE REFERENCE

8.1 Building a VM

Last updated 3 February 2014 for Self 4.5.0

The sources for the Self VM are on the [GitHub repository](#).

8.1.1 Requirements

On Mac OS X you will need XCode, version 4 or higher, with the command line tools installed and CMake in at least version 2.8. CMake can be easily installed using the Brew package manager.

On Linux you will generally need the appropriate libraries for Git, CMake, a GCC or Clang toolchain, X and Ncurses. For example, on Fedora 19 you will need: `git cmake gcc gcc-c++ glibc-devel.i686 libstdc++-devel libstdc++-devel.i686 libX11-devel.i686 libXt-devel.i686 libXext-devel.i686 libXrender-devel.i686 libXau-devel.i686 libxcb-devel.i686 ncurses-devel.i686 ncurses-libs.i686`

Self builds with GCC 4.2 or Clang 2.0.

8.1.2 Building

You can do an in-tree build with:

```
cmake .
cmake --build .
```

or an out-of-tree build with:

```
mkdir -p $YOU_BUILD_DIRECTORY; cd $YOU_BUILD_DIRECTORY
cmake $DIRECTORY_OF_SELF_CHECKOUT
cmake --build $YOU_BUILD_DIRECTORY
```

CMake respects your environment variables, so to build Self with Clang, configure it like this:

```
CC=clang CXX=clang++ cmake $DIRECTORY_OF_SELF_CHECKOUT
```

On Mac OS X with Xcode, you can use the Xcode generator of CMake like this:

```
cmake -GXcode $DIRECTORY_OF_SELF_CHECKOUT
```

On 64bit Linux, you may want to explicitly use 32bit compilation:

```
CC="gcc -m32" CXX="g++ -m32" cmake $DIRECTORY_OF_SELF_CHECKOUT
```

The same holds for Clang.

You may wish to use `ccmake` or the CMake GUI (`cmake-gui` or the CMake.app on OS X) to configure available features like Release/Debug/Profiled builds or to enable experimental features.

8.2 Startup options

The following command-line options are recognised by the Virtual Machine:

Argument	Description
<code>-f filename</code>	Reads filename (which should contain Self source) immediately after startup (after reading the snapshot) and evaluates the contents. Useful for setting options, installing personal short-cuts, etc.
<code>-h</code>	Prints a message describing the options.
<code>-p</code>	Suppresses execution of the expression <code>snapshotAction postRead</code> after reading a snapshot. Useful if something in the startup sequence causes the system to break.
<code>-s snapshot</code>	Reads initial world from snapshot. A snapshot begins with the line <code>exec Self -s \$0 \$@</code> which causes the Virtual Machine to begin execution with the snapshot.
<code>-w</code>	Don't print warnings about object code.

These options are provided for use by Self VM implementors:

Argument	Description
<code>-F</code>	Discards any machine code saved in the snapshot. If the code in a snapshot is for some reason corrupted, but the objects are not, this option can be used to recover the snapshot.
<code>-l logfile</code>	Writes a log of events generated by the spy to logfile.
<code>-r</code>	Disables real timer interrupts.
<code>-t</code>	Disables all timers.

Other command-line options are ignored by the Virtual Machine but are available at Self level via the primitive `_CommandLine`.

The standard set of Self objects (built by the `worldBuilder.self` script) also defines `-b` (where the objects director is) and `-o` (for specifying build options)

8.3 System-triggered messages

Certain events cause the system to automatically send a message to the lobby. After reading a snapshot the expression `snapshotAction postRead` is evaluated. This allows the Self world to reinitialize itself — for example, to reopen windows.

There are other situations in which the system sends messages; see *Run-time message lookup errors*.

8.4 Run-time message lookup errors

If an error occurs during a message send, the system sends a message to the receiver of the message. Any object can handle these errors by defining (or inheriting) a slot with the corresponding selector. All messages sent by the system in response to a message lookup error have the same arguments. The first argument is the offending message's selector; the additional arguments specify the message send type (one of `'normal'`, `'implicitSelf'`,

'undirectedResend', 'directedResend', or 'delegated'), the directed resend parent name or the delegatee (0 if not applicable), the sending method holder, and a vector containing the arguments to the message, if any.

undefinedSelector:Type:Delegatee:MethodHolder:Arguments

The receiver does not understand the `message:` no slot matching the selector can be found in the receiver or its ancestors.

ambiguousSelector:Type:Delegatee:MethodHolder:Arguments

There is more than one slot matching the selector.

missingParentSelector:Type:Delegatee:MethodHolder:Arguments

The parent slot through which the resend should have been directed was not found in the sending method holder.

mismatchedArgumentCountSelector:Type:Delegatee:MethodHolder:Arguments

The number of arguments supplied to the `_Perform` primitive does not match the number of arguments required by the selector.

performTypeErrorSelector:Type:Delegatee:MethodHolder:Arguments

The first argument to the `_Perform` primitive (the selector) wasn't a canonical string.

These error messages are just like any other message. Therefore, it is possible that the object *P* causing the error (which is being sent the appropriate error message) does not understand the error message *M* either. If this happens, the system sends the first message (`undefinedSelector:`) to the current process, with the error message *M* as argument. If this is not understood, then the system suspends the process. If the scheduler is running, it is notified of the failure.

The system will also suspend a process if it runs out of stack space (too much recursion) or if a block is evaluated whose lexically-enclosing scope has already returned. Since these errors are nonrecoverable they cannot be caught by the same Self process; the scheduler, if running, is notified.

8.5 Low-level error messages

Five kinds of errors can occur during the execution of a Self program: lookup errors, primitive errors, programmer defined errors, non-recoverable errors, and fatal VM errors. All but the last of these are usually caught and handled by mechanisms in the programming environment, resulting in a debugger being presented to the user. However, if programs are run without the programming environment, or the error-handling mechanisms themselves are broken, low-level error facilities are used.

This section describes the various error messages presented by the low-level facilities. For each category or error, the general layout of error messages in that category will be explained along with the format of the stack trace. Then a “rogue’s gallery” of the errors in that category will be shown.

By default, errors are handled by a set of methods defined in module `errorHandling`. For all errors except nonrecoverable and fatal VM errors, an object can handle errors in its own way by defining its own error handling methods. If the object in which an error occurs neither inherits nor defines error handling behavior, the VM prints out a low-level error message and a stack trace. The system will also resort to this low-level message and trace if an error is encountered while trying to handle an error.

8.6 An example

Here is an expression that produces an error in the current system:

```
"Self 7" 100000 factorial
The stack has grown too big.
(Self limits stack sizes, and cannot resume processes with stack overflows.)
To debug type "attach" or to show stack type "zombies first printError".
```

The error arose because the recursive method `factorial` exceeded the size allocated for the process stack which resulted in a stack overflow.

The virtual machine currently allocates a fixed-size stack to each process and does not extend the stack on demand.

8.7 Lookup errors

Lookup errors occur when an object does not understand a message that is sent to it. How the actual message lookup is done is described in the *Language Reference* chapter.

No 'foo' slot found in shell <0>.

The lookup found no slot matching the selector `foo`.

No 'fish' delegatee slot was found in <a child of lobby> <12>.

The lookup found no parent slot `fish`, which was explicitly specified as the delegatee of the message.

More than one 'system' slot was found in shell <0>.

The matching slots are: oddballs <6> and prototypes <7>.

The lookup found two matching `system` slots which means the message is ambiguous. The error message also says where the matching slots were found.

8.8 Programmer defined errors

These are explicitly raised in the Self program to report errors, e.g. sending the message `first` to an empty list will cause such an error.

```
Error: first is absent.
Receiver is: list <7>.
```

Use the selectors `error:` and `error:Arguments:` to raise a programmer defined error.

8.9 Primitive errors

Primitive failures occur when a primitive cannot perform the requested operation, for example, because of a missing or invalid argument.

```
badTypeError: the '_IntAdd:' primitive failed.
Its receiver was shell <6>.
```

The primitive failed with `badTypeError` because the shell is not an integer.

```
The selector 12 could not be sent to shell because it is not a string.
```

The primitive `_Perform` expects a string as its first argument.

```
The selector 'add:' could not be sent to shell <0> because it does not take 2_
↳arguments.
```

The primitive `_Perform` received the wrong number of arguments.

There are many other kinds of possible primitive errors.

8.10 Nonrecoverable process errors

Errors that stop a process from continuing execution are referred to as nonrecoverable errors.

```
The stack has grown too big.
(Self 4.0 limits stack sizes, and cannot resume processes with stack
overflows.)
```

A stack overflow error occurs because the current version of Self allocates a fixed size stack for each process, and the stack cannot be expanded.

```
Self 4.0 cannot run a block after its enclosing method has returned.
(Self cannot resume this process, either.)
```

This error occurs if a block is executed after its lexically enclosing method has returned. This is called a “non-LIFO” block. Non-LIFO blocks are not supported by the current version of Self.

8.11 Fatal errors

In rare cases, the virtual machine may encounter a fatal error (e.g., a resource limit is exceeded or an internal error is discovered). When this happens, a short menu is displayed:

```
VM Version: 4.0.5, Tue 27 Jun 95 13:35:49 Solaris 2.x (svr4)
Internal error: signal 11 code 3 addr 4 pc 0x1ac768.
Do you want to:
1) Quit Self (optionally attempting to write a snapshot)
2) Try to print the Self stack
3) Try to return to the Self prompt
4) Force a core dump
Your choice:
```

The first two lines help the Self implementors locate the problem. Printing the Self stack may provide more information about the problem but does not always work. Returning to the Self prompt may be successful, but the system integrity may have been compromised as a result of the error. The safest course is to attempt to write a snapshot (if there are unsaved changes), and then check the integrity of the snapshot by executing the primitive `_Verify` after starting it. If there are any error messages from the primitive, do not attempt to continue using the snapshot.

Since fatal errors usually arise from a bug in the virtual machine, please send the Self group a bug report, and include a copy of the error message if possible. If the error is reproducible please describe how to reproduce it (including a snapshot or source files may be helpful).

8.12 The initial Self world

The diagram on the following pages shows all objects in the “bare” Self world. In addition, literals like integers, floats, and strings are conceptually part of the initial Self world; block and object literals are created by the programmer as needed. All the objects in the system are created by adding slots to these objects or by cloning them. Table 8.1 lists all the initial objects and provides a short description for each. Reading in the world rearranges the structure of the “bare” Self world (see *The Self World*).

Table 8.1: Objects in the initial Self world

Object	Description
lobby	The center of the Self object hierarchy, and the context in which expressions typed in at the VM prompt, read in via <code>_RunScript</code> , or used as the initializers of slots, are evaluated.

Table 8.2: Objects in the lobby

Object	Description
shell	After reading in the world, shell is the context in which expressions typed in at the prompt are evaluated.
snapshotAction	An object with slot for the startup action (see <i>System-triggered messages</i>), <code>postRead</code> . This slot initially contains <code>nil</code> .
systemObjects	This object contains slots containing the general system objects, including <code>nil</code> , <code>true</code> , <code>false</code> , and the prototypical vectors and mirrors.

Table 8.3: Objects in systemObjects

Object	Description
<code>nil</code>	The initializer for slots that are not explicitly initialized. Indicates “not a useful object.”
<code>true</code>	Boolean <code>true</code> . Argument to and returned by some primitives.
<code>false</code>	Boolean <code>false</code> . Argument to and returned by some primitives.
<code>vector</code>	The prototype for (normal) vectors.
<code>byteVector</code>	The prototype for byte vectors.
<code>proxy</code>	The prototype for proxy objects.
<code>fctProxy</code>	The prototype for <code>fctProxy</code> objects.
<code>vector parent</code>	The object that <code>vector</code> inherits from. Since all object vectors will inherit from this object (because they are cloned from <code>vector</code>), this object will be the repository for shared behavior (a traits object) for vectors.
<code>byteVector parent</code>	Similar to <code>vector parent</code> : the <code>byteVector</code> traits object.
<code>slotAnnotation</code>	The default slot annotation object.
<code>objectAnnotation</code>	The default object annotation object
<code>profiler</code>	The prototype for profilers.
<code>mirrors</code>	See below.

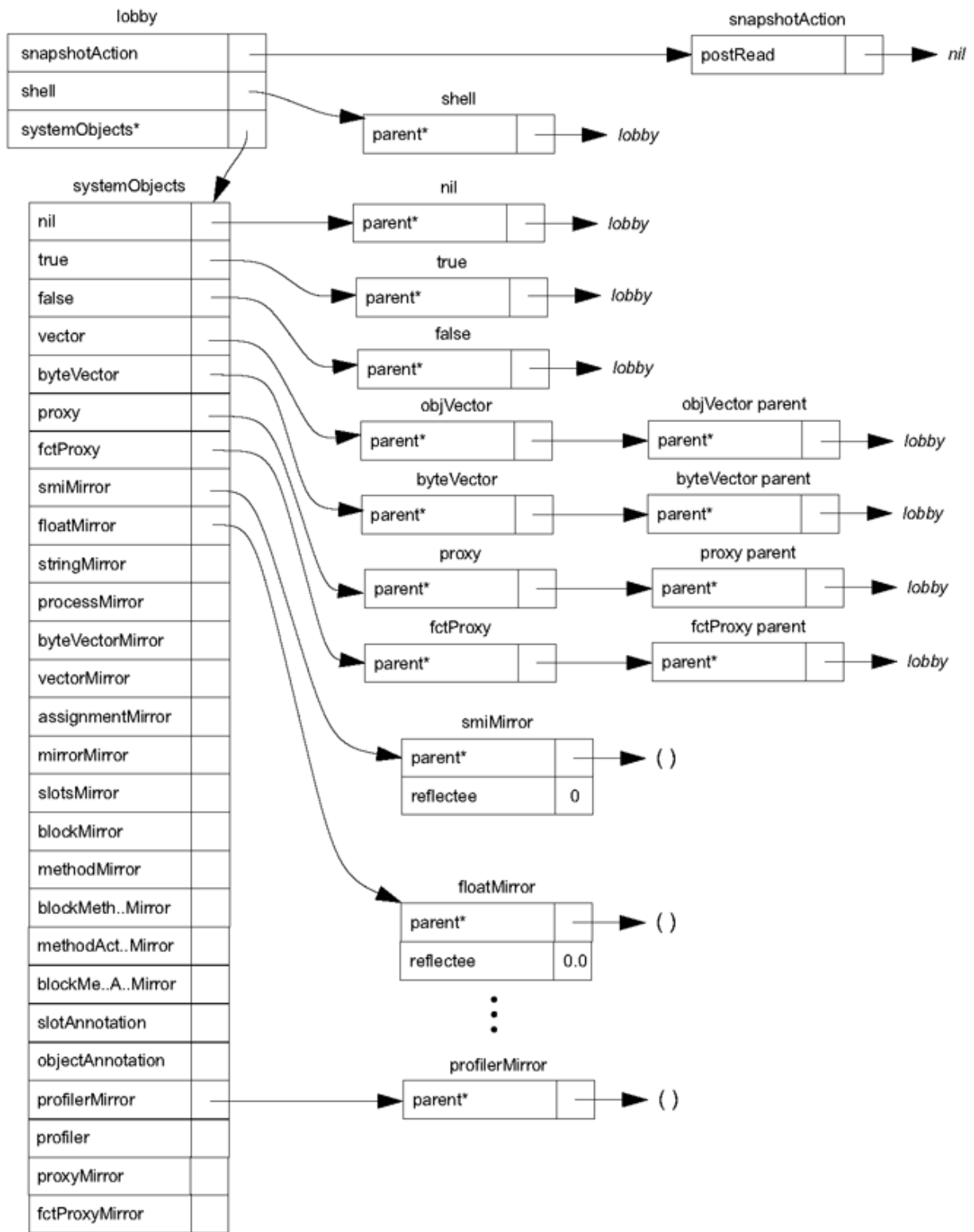


Fig. 8.1: The initial Self world (part 1)

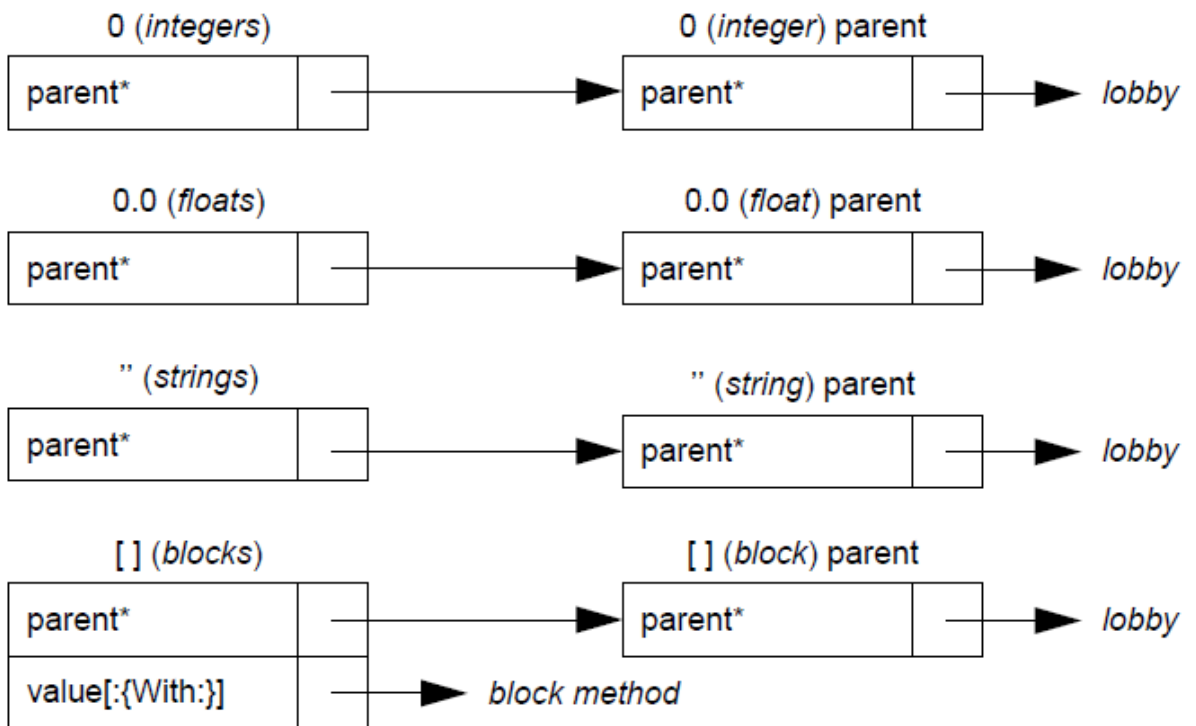


Fig. 8.2: The initial Self world (part 2)

Table 8.4: Literals and their parents

Literal	Description
integers	Integers have one slot, a parent slot called <code>parent</code> . All integers have the same parent: see <code>0 parent</code> , below.
<code>0 parent</code>	All integers share this parent, the integer traits object.
floats	Floats have one slot, a parent slot called <code>parent</code> . All floats have the same parent: see <code>0.0 parent</code> , below.
<code>0.0 parent</code>	All floats share this parent, the float traits object.
canonical strings	In addition to a byte vector part, a canonical string has one slot, <code>parent</code> , a parent slot containing the same object for all canonical strings (see <code>" parent</code> below).
<code>' ' parent</code>	All canonical strings share this parent, the string traits object.
blocks	Blocks have two slots: <code>parent</code> , a parent slot containing the same object for all blocks (see <code>[] parent</code> , below), and <code>value</code> (or <code>value:</code> , or <code>value:With:</code> , etc., depending on the number of arguments the block takes) which contains the block's deferred method.
<code>[] parent</code>	All blocks share this parent, the block traits object.

Table 8.5: Prototypical mirrors

Mirror	Description
<code>smiMirror</code>	Prototypical mirror on a small integer; the reflectee is <code>0</code> .
<code>floatMirror</code>	Prototypical mirror on a float; the reflectee is <code>0.0</code> .
<code>stringMirror</code>	Prototypical mirror on a canonical string; the reflectee is the empty canonical string (<code>"</code>).
<code>processMirror</code>	Prototypical mirror on a process; the reflectee is the initial process.
<code>byteVectorMirror</code>	Prototypical mirror on a byte vector; the reflectee is the prototypical byte vector.
<code>objVectorMirror</code>	Prototypical mirror on object vectors; the reflectee is the prototypical object vector.
<code>assignmentMirror</code>	Mirror on the assignment primitive; the actual reflectee is an empty object.
<code>mirrorMirror</code>	Prototypical mirror on a mirror; the reflectee is <code>slotsMirror</code> .
<code>slotsMirror</code>	Prototypical mirror on a plain object without code; the reflectee is an empty object.
<code>blockMirror</code>	Prototypical mirror on a block.
<code>methodMirror</code>	Prototypical mirror on a normal method.
<code>blockMethodMirror</code>	Prototypical mirror on a block method.
<code>methodActivationMirror</code>	Prototypical mirror on a method activation.
<code>blockMethodActivationMirror</code>	Prototypical mirror on a block activation.
<code>proxyMirror</code>	Prototypical mirror on a proxy.
<code>fctProxyMirror</code>	Prototypical mirror on a <code>fctProxy</code> .
<code>profilerMirror</code>	Prototypical mirror on a profiler.

All of the prototypical mirrors consist of one slot, a parent slot named `parent`. Each of these parent slots points to an empty object (denoted in Fig. 8.1 by `()`).

8.13 Option Primitives

This section has not been updated to include all options present in Self 4.0.

Option primitives control various aspects of the Self system and its inner workings. Many of them are used to debug or instrument the Self system and are probably of little interest to users. The options most useful for users are listed in Table 8.6; other option primitives can be found in Appendix 10.8 *Primitives*, and a list of all option primitives and their current settings can be printed with the primitive `_PrintOptionPrimitives`.

Table 8.6: Some useful option primitives

Name	Description
<code>_PrintPeriod[:]</code>	Print a period when reading a script file with <code>_RunScript</code> . Default: <code>false</code> .
<code>_PrintScriptName[:]</code>	Print the file name when reading a script file. Default: <code>false</code> .
<code>_Spy[:]</code>	Start the system monitor (see Appendix 10.7 for details). Default: <code>false</code> .
<code>_StackPrintLimit[:]</code>	Controls the number of stack frames printed by <code>_PrintProcessStack</code> . Default: 20.
<code>_DirPath[:]</code>	The default directory path for script files.

Each option primitive controls a variable within the virtual machine containing a boolean, integer, or string (in fact, the option primitives can be thought of as “primitive variables”). Invoking the version of the primitive that doesn’t take an argument¹ returns the current setting; invoking it with an argument sets the variable to the new value and returns the old value.

Try running the system monitor with `_Spy: true`. The system monitor will continuously display various information about the system’s activities and your memory usage.

¹ The bracketed colon indicates that the argument is optional (i.e., there are two versions of the primitive, one taking an argument and one not taking an argument). The bracket is not part of the primitive name. See text for details.

8.14 Interfacing with other languages

This chapter describes how to access objects and call routines that are written in other languages than Self. We will refer to such entities as *foreign objects* and *foreign routines*. A typical use would be to make a function found in a C library accessible in Self. Three steps are necessary to accomplish this:

- Write and compile a piece of “glue” code that specifies argument and result types for the foreign routine and how to convert between these types and Self objects.
- Link the resulting object code to the Self virtual machine.
- Create a function proxy object (actually a `foreignFct` object) that represents the routine in the Self world.

Each of these steps is described in detail in the following sections.

8.14.1 proxy and fctProxy objects

A foreign object is represented by a proxy object in the Self world. A *proxy* object is an object that encapsulates a pointer to the foreign object it represents. In addition to the pointer to the foreign object, the proxy object contains a type seal. A type seal is an immutable value that is assigned to the proxy object, when it is created. The *type seal* is intended to capture type information about the pointer encapsulated in the proxy. For example, proxies representing window objects should have a different type seal than proxies representing event objects. By checking the type seal against an expected value whenever a proxy is “opened”, many type errors can be caught. The last property of proxy objects is that they can be *dead* or *live*. If an attempt is made to use the pointer in a dead proxy object, an error results (`deadProxyError`). Proxy objects may be explicitly killed, by sending the primitive message `_Kill` to them. Furthermore, they are automatically killed after reading in a snapshot. This way problems with dangling references to foreign objects that were not included in the snapshot are avoided.

`fctProxy` objects are similar to proxy objects: they have a type seal and are either live or dead. However, they represent a foreign routine, rather than a foreign object. A foreign routine can be invoked by sending the primitive messages `_Call`, `_Call:{With:}`, `_CallAndConvert{With:And:}` to the `fctProxy` representing it. Note that `fctProxy` objects are low-level. Most, if not all, uses of foreign routines should use the interface provided by `foreignFct` objects.

Proxies (and `fctProxies`) can be freely cloned. However a cloned proxy will be dead. A dead proxy is revived when it is used by a foreign function to, e.g., return a pointer. The return value of the foreign function together with a type seal is stored into the dead proxy, which is then revived and returned as the result of the foreign routine call. The motivation for this somewhat complicated approach is that there will be several different kinds of proxies in a typical Self system. Different kinds of proxies may have different slots added, so rather than having the foreign routine figure out which kind of proxy to clone for the result, the Self code calling the foreign routine must construct and pass down an “empty” (dead) proxy to hold the result. This proxy is called a *result proxy* and it is the last argument supplied to the foreign function.

8.14.2 Glue code

Glue code is responsible for the transition from Self to foreign routines. It forms wrappers around foreign routines. There is one wrapper per foreign routine. A wrapper takes a number of arguments of type `oop`, and returns an `oop` (`oop` is the C++ type for “reference to Self object”). When a wrapper is executed, it performs the following steps:

1. Check that the arguments supplied have the correct types.
2. Convert the arguments from Self representation to the representation that the foreign routine needs.
3. Invoke the foreign routine on the converted arguments.
4. Convert the return value of the foreign routine to a Self object and return this as the Self level result.

To make it easier to write glue code, a special purpose language has been designed for this. The result is that glue for a foreign routine will often consist of only a single line. The glue language is implemented as a set of C++ preprocessor macros. Therefore, glue code is just a (rather peculiar) kind of C++. Glue code can be in a file of its own, or – if it is glue for calling C++ routines – it can be in the same file as the foreign routines, and compiled with them.

To make the definition of the glue language available, the file containing glue code must contain:

```
# include "_glueDefs.c.incl"
```

The file “_glueDefs.c.incl” includes a bunch of C++ header files that contain all the definitions necessary for the glue. Of the included files, “glueDefs.h” is probably the most interesting in this context. It defines the glue language and also contains some comments explaining it.

Since different foreign languages have different type systems and calling conventions the glue language is actually not a single language, but one for each supported foreign language. Presently C and C++ are supported. See sections *C glue* and *C++ glue* for details.

8.14.3 Compiling and linking glue code

Since glue code is a special form of C++ code, a C++ compiler is needed to translate it. The way this is done may depend on the computer system and the available C++ compiler. The following description applies to Sun SPARCstations using the GNU g++ compiler.

A specific example of how to compile glue code can be found in the directory containing the *toself* demo (see *A complete application using foreign functions* for further details). The makefile in that directory describes how to translate a .c file containing glue into something that can be invoked from Self. This is a two stage process: first the .c file is compiled into a .o file which is then linked (perhaps with other .o files and libraries that the glue code depends on) into a .so file (a so-called dynamic library). While the compilation is straightforward, several issues concerning the linking must be explained.

Linking Before a foreign routine can be called it must be linked to the Self virtual machine. The linking can be done either statically, i.e. before Self is started, or dynamically, i.e. while Self is running. The Self system employs both dynamic and static linking, but users should only use dynamic linking, as static linking requires more understanding of the structure of the Virtual Machine. The choice between dynamic and static linking involves a trade-off between safety and flexibility as outlined in the following.

Dynamic linking Dynamic linking has the advantage that it is done on demand, so only foreign routines that are actually used in a particular session will be loaded and take up space. Debugging foreign routines is also easier, especially if the dynamic linker supports unlinking. The main disadvantages with dynamic linking is that more things can go wrong at run time. For example, if an object file containing a foreign routine can not be found, a run time error occurs. The Sun OS dynamic linker, ld.so, only handles dynamic libraries which explains why the second stage of glue translation is necessary.

Static linking Static linking, the alternative that was not chosen for Self, has the advantage that it needs to be done only once. The statically linked-in files will then be available for ever after. The main disadvantages are that the linked-in files will always take up space whether used or not in a given Self session, that the VM must be completely relinked every time new code is added, and that debugging is harder because there is no way to unlink code with bugs in. For these reasons the following examples all use dynamic linking.

8.14.4 A simple glue example: calling a C function

Suppose we have a C function that encrypts text strings in some fancy way. It takes two arguments, a string to encrypt and a key, and returns a string which is the result of the encryption. To use this function from Self, we write a line of C glue. Here is the entire file, “encrypt.c”, containing both the encryption function and the glue:

```

/* Make glue available by including it. */
# include "incls/_glueDefs.c.incl"
/* Naive encryption function. */
char *encrypt(char *str, int key) {
    static char res[1000];
    int i;
    for (i = 0; str[i]; ++i)
        res[i] = str[i] + key;
    res[i] = '\0';
    return res;
}

/* Make glue expand to full functions, not just prototypes. */
# define WHAT_GLUE FUNCTIONS
    C_func_2(string,, encrypt, encrypt_glue,, string,, int,)
# undef WHAT_GLUE

```

A few words of explanation: the last three lines of this file contain the glue code. First defining `WHAT_GLUE` to be `FUNCTIONS`, makes the following line expand into a full wrapper function (defining `WHAT_GLUE` to be `PROTOTYPES` instead, will cause the `C_func_2` line to produce a function prototype only). The line containing the macro `C_func_2` is the actual wrapper for `encrypt`. The “2” designates that `encrypt` takes 2 arguments. The meaning of the arguments, from left to right are:

- `string,:` specifies that `encrypt` returns a string argument.
- `encrypt:` name of function we are constructing wrapper for.
- `encrypt_glue:` name that we want the wrapper function to have.
- An empty argument signifying that `encrypt` is not to be passed a failure handle (explained later).
- `string,:` specifies that the first argument to `encrypt` is a string.
- `int,:` specifies that the second argument to `encrypt` is an int.

Having written this file, we now prepare a makefile to compile and link it. To do this, we can extend the makefile in `objects/glue/{sun4,svr4}` (depending on OS in use) and then run `make`. This results in the shared library file `encrypt.so`. Finally, to try it out, we can type these commands (at the Self prompt or in the UI):

```

> _AddSlotsIfAbsent: ( | encrypt | )
lobby

> encrypt: ( foreignFct copyName: 'encrypt_glue' Path: 'encrypt.so' )
lobby

> encrypt
<C++ function(encrypt_glue)>

> encrypt value: 'Hello Self' With: 3
'Khoor#Vhoi'

> encrypt value: 'Khoor#Vhoi' With: -3
'Hello Self'

```

Comparing the signature for the function `encrypt` with the arguments to the `C_func_2` macro it is clear that there is a straightforward mapping between the two. One day we hope to find the time to write a Self program that can parse a C or C++ header file and generate glue code corresponding to the definitions in it. In the meantime, glue code must be handwritten.

8.14.5 C glue

C glue supports accessing C functions and data from Self. There are three main parts of C glue:

- Calling functions.
- Reading/assigning global variables.
- Reading/assigning a component in a struct that is represented by a proxy object in Self.

In addition, C++ glue for creating objects can be used to create C structs (see section *C++ glue*). The following sections describe each of these parts of C glue.

8.14.6 Calling C functions

The macro `C_func_N` where `N` is 0, 1, 2, ... is used to “glue in” a C function. The number `N` denotes the number of arguments that should be given *at the Self level*, when calling the function. This number may be different from the number of arguments that the C function takes since, e.g., some argument conversions (see below) produce two C arguments from one Self object. Here is the general syntax for `C_func_N`:

```
C_func_N(res_cnv, res_aux, fexp, gfname, fail_opt, c0, a0, ... cN, aN)
```

Compare this with the glue that was used in the `encrypt` example in section *A simple glue example: calling a C function*:

```
C_func_2(string,, encrypt, encrypt_glue,, string,, int,)
```

The meaning of each argument to `C_func_N` is as follows:

- `res_cnv, res_aux`: these two arguments form a “conversion pair” that specifies how the result that the function returns is converted to a Self object. In the `encrypt` example, where the function returns a null terminated string, `res_cnv` has the value `string`, and `res_aux` is empty. Table 8.7 lists all the possible values for the `res_cnv, res_aux` pair.
- `fexp` is a C expression which evaluates to the function that is being glued in. In the simplest case, such as in the `encrypt` example, the expression is the name of a function, but in general it may be any C expression, involving function pointers etc., which in a global context evaluates to a function.
- `gfname`: the name of the function which the `C_func_N` macro expands into. In the `encrypt` example, the convention of appending `_glue` to the C function’s name was used. When accessing a glued-in function from Self, the value of `gfname` is the name that must be used.
- `fail_opt`: there are two possible values for this argument. It can be empty (as in the example) or it can be `fail`. In the latter case, the C function being called is passed an additional argument that will be the last argument and have type `void *`. Using this argument, the C function may abort its execution and raise an exception. The result is that the “IfFail block” in Self will be invoked.
- `ci, ai`: each of these pairs describes how to convert a Self level argument to one or more C level arguments. For example, in the glue for `encrypt`, `c0, “a0”` specifies that the first argument to `encrypt` is a string. Likewise `c1, “a1”` specifies that the second argument is an integer. Note that in both these cases, the a-part of the conversion is empty. Table 8.7 lists all the possible values for the `ci, “ai”` pair.

Handling failures. Here is a slight modification of the encryption example to illustrate how the C function can raise an exception that causes the “IfFail block” to be invoked at the Self level:

```
/* Make glue available by including it. */
# include "incls/_glueDefs.c.incl"
/* Naive encryption function. */
char *encrypt(char *str, int key, void *FH) {
```

```

static char res[1000];
int i;
if (key == 0) {
    failure(FH, "key == 0 is identity map");
    return NULL;
}
for (i = 0; str[i]; i++)
    res[i] = str[i] + key;
res[i] = '\0';
return res;
}
/* Make glue expand to full functions, not just prototypes. */
# define WHAT_GLUE FUNCTIONS
    C_func_2(string,, encrypt, encrypt_glue, fail, string,, int,)
# undef WHAT_GLUE

```

Observe that the `fail_opt` argument now has the value `fail` and that the `encrypt` function raises an exception, using `failure`, if the key is 0. There are two ways to raise exceptions:

```

extern "C" void failure(void *FH, char *msg);
extern "C" void unix_failure(void *FH, int err = -1);

```

In both cases, the `FH` argument is the “failure handle” that was passed by the `C_func_N` macro. The second argument to `failure` is a string. It will be passed to the “IfFail block” in Self. `unix_failure` takes an optional integer as its second argument. If this integer has the value `-1`, or is missing, the value of `errno` is used instead. The integer is interpreted as a UNIX error number, from which a corresponding string is constructed. The string is then, as for `failure`, passed to the “IfFail block” at the call site in Self.

Warning: After calling `failure` or `unix_failure` a normal return must be done. The value returned (in the example `NULL`) is ignored.

8.14.7 Reading and assigning global variables

Reading the value of a global variable is done using the `C_get_var` macro. Assigning a value to a global variable is done using `C_set_var`. Both macros expand into a C++ function that converts between Self and C representation, and reads or assigns the variable. Here is the general syntax:

```

C_get_var(cnvt_res,aux_res, expr, gfname)
C_set_var(var, expr_c0,expr_a0, gfname)

```

A concrete example is reading the value of the variable `errno`, which can be done using:

```

C_get_var(int,, errno, get_errno_glue)

```

The meaning of the each argument is:

- `cnvt_res`, “`aux_res`“: how to convert the value of the global variable that is being read to a Self object. In the `errno` example, `cnvt_res` is `int` and `aux_res` is empty, since the type of `errno` is `int`. The `cnvt_res`, “`aux_res`“ can be any one of the result conversions found in [Table 8.7](#).
- `expr` is the variable whose value is being read. In the `errno` example, it is simply `errno`, but in general, it may actually be any expression that is valid in a global context, even an expression involving function calls.
- `gfname`: the name of the C++ function that `C_get_var` or `C_set_var` expands into.

- `var` is the name of a global variable that a value is assigned to. In general, `var`, may be any expression that in a global context evaluates to an l-value.
- `expr_c0`, “`expr_a0`“: when assigning to a variable, the value it is assigned is obtained by converting a Self object to a C value. The `expr_c0`, “`expr_a0`“ pair, which can be any one of the argument conversions listed in Table 8.7, specifies how to do this conversion.

8.14.8 Reading and assigning struct components

Reading the value of a struct component or assigning a value to it is similar to doing the same operations on a global variable. The difference is that the struct must somehow be specified. This is taken care of by the macros `C_get_comp` and `C_set_comp`. The general syntax is:

```
C_get_comp(cnvt_res,aux_res, cnvt_strc,aux_strc, comp, gfname)
C_set_comp(cnvt_strc,aux_strc, comp, expr_c0,expr_a0, gfname)
```

Here is an example, assigning to the `sin_port` field of a struct `sockaddr_in` (this struct is defined in `/usr/include/netinet/in.h`):

```
struct sockaddr_in {
    short          sin_family;
    u_short       sin_port;
    struct in_addr sin_addr;
    char          sin_zero[8];
};
```

The struct is represented by a proxy object:

```
char *socks = "type seal for sockaddr_in proxies";
C_set_comp(proxy, (sockaddr_in *,socks), .sin_port, short,,set_sin_port_glue)
```

The `sockaddr_in` example defines a function, `set_sin_port_glue`, which can be called from Self. The function takes two arguments, the first being a proxy representing a `sockaddr_in` struct, the second being a short integer. After converting types, `set_sin_port_glue` performs the assignment:

```
(*first_converted_arg).sin_port = second_converted_arg.
```

In general the meaning of the `C_get_comp` and `C_set_comp` arguments is:

- `cnvt_res`, `aux_res`: how to convert the value of the component that is being read to a Self object. Any of the result conversions found in Table 8.7 may be applied.
- `cnvt_strc`, `aux_strc`: the conversion that is applied to produce a struct upon which the operation is performed. In the `sin_port` example, this conversion is a proxy conversion, implying that in Self, the struct whose `sin_port` component is assigned is represented by a proxy object. In general, any of the argument conversions from Table 8.7 that results in a pointer, may be used.
- `comp` is the name of the component to be read or assigned. In the `sin_port` example, this name is “`.sin_port`”. Note that it includes a “.”. This, e.g., allows handling pointers to int’s by pretending that it is a pointer to a struct and operating on a component with an empty name.
- `gfname`: the name of the C++ function that `C_get_comp` or `C_set_comp` expands into.
- `expr_co`, `expr_a0`: when assigning to a component, the value it is assigned is obtained by converting a Self object to a C value. The `expr_co`, `expr_a0` pair, which can be any one of the argument conversions listed in Table 8.7, specifies how to do this conversion.

8.14.9 C++ glue

Since C++ is a superset of C, all of C glue can be used with C++. In addition, C++ glue provides support for:

- Constructing objects using the new operator.
- Deleting objects using the delete operator.
- Calling member functions on objects.

Each of these parts will be explained in the following sections.

8.14.10 Constructing objects

In C++, objects are constructed using the new operator. Constructors may take arguments. The macros `CC_new_N` where `N` is a small integer, support calling constructors with or without arguments. Calling a constructor is similar to calling a function, so for additional explanation, please refer to section *Calling C functions*. Here is the general syntax for constructing objects using C++ glue:

```
CC_new_N(cnvt_res,aux_res, class, gfname, c0,a0, c1,a1, ... cN,aN)
```

For example, to construct a `sockaddr_in` object, the following glue statement could be used:

```
CC_new_0(proxy, (sockaddr_in *,socks), sockaddr_in, new_sockaddr_in)
```

The meanings of the `CC_new_N` arguments are as follows:

- `cnvt_res, aux_res`: the result of calling the constructor is an object pointer. The result conversion pair `cnvt_res, aux_res` (see [Table 8.7](#)), specifies how this pointer is converted to a Self object before being returned. In the `sockaddr` example, the proxy result conversion is used.
- `class` is the name of the class (or struct) that is being instantiated.
- `gfname`: the name of the C++ function that the `CC_new_N` macro expands into.
- `ci, ai`: if the constructor takes arguments, these arguments must be converted from Self representation to C++ representation. The arguments conversion pairs `ci, ai` specify how each argument is converted. See [Table 8.7](#) for a description of all argument conversions. In the `sockaddr` example, there are no arguments.

8.14.11 Deleting objects

C++ objects can have destructors that are executed when the objects are deleted. To ensure that the destructor is called properly, the `delete` operator must know the type of the object being deleted. This is ensured by using the `CC_delete` macro, which has the following form:

```
CC_delete(cnvt_obj,aux_obj, gfname)
```

For example, to delete `sockaddr_in` objects (constructed as in the previous section), the `CC_delete` macro should be used in this manner:

```
CC_delete(proxy, (sockaddr_in *,socks), delete_sockaddr_in)
```

In general, the meaning of the arguments given to `CC_delete` is:

- `cnvt_obj, aux_obj`: this pair can be any of the argument conversions found in [Table 8.7](#) that produces a pointer to the object that will be deleted.
- `gfname`: the name of the C++ function that this invocation of `CC_delete` expands into.

8.14.12 Calling member functions

Table 8.7 lists all the available argument conversions. Each row represents one conversion, with the first two columns designating the conversion pair. The third column lists the types of Self objects that the conversion pair accepts. The fourth column lists the C types that it produces. The fifth column lists the kind of errors that can occur during the conversion. Finally, the sixth column contains references to numbered notes. The notes are found in the paragraphs following the table.

Calling member functions is similar to calling “plain” functions, so please also refer to section *Calling C functions*. The difference is that an additional object must be specified: the object upon which the member function is invoked (the receiver in Self terms). Calling a member function is accomplished using one of the macros:

```
CC_mber_N(cnvt_res,aux_res, cnvt_rec,aux_rec, mname, gfname,
          fail_opt, c0,a0, c1,a1, ..., cN,aN)
```

For example here is how to call the member function `zock` on a `sockaddr_in` object given by a proxy:

```
CC_mber_0(bool,, proxy,(sockaddr_in *,socks), zock, zock_glue,)
```

The arguments to `CC_mber_N` are:

- `cnvt_res, aux_res`: this pair, which can be any of the result conversions from Table 8.7, specifies how to convert the result of the member function before returning it to Self. For example, the `zock` member function returns a boolean.
- `cnvt_rec, aux_rec`: the object on which the member function is invoked. Often this will be a proxy conversion as in the `zock` example.
- `mname` is the name of the member function. In general, it may be any expression, such that `receiver->mname` evaluates to a function.
- `gfname` is the name of the C++ function that the `CC_mber_N` macro expands into.
- `fail_opt`: whether or not to pass a failure handle to the member function (refer to section *Calling C functions* for details).
- `ci, ai`: these are argument conversion pairs specifying how to obtain the arguments for the member function. Any conversion pair found in Table 8.7 may be used.

8.14.13 Conversion pairs

A major function of glue code is to convert between Self objects and C/C++ values. This conversion is guarded by so-called conversion pairs. A *conversion pair* is a pair of arguments given to a glue macro. It handles converting one or at most a few types of objects/values. There are different conversion pairs for converting from Self objects to C/C++ values (called argument conversion pairs) and for converting from C/C++ values to Self objects (called result conversion pairs).

8.14.14 Argument conversions – from Self to C/C++

An argument conversion is given a Self object and performs these actions to produce a corresponding C or C++ value:

- check that the Self object it has been given is among the allowed types. If not, report `badTypeError` (invoke the failure block (if present) with the argument `'badTypeError'`).
- check that the object can be converted to a C/C++ value without overflow or any other error. If not, report the relevant error.
- do the conversion, i.e., construct the C/C++ value corresponding to the given Self object.

Table 8.7: Argument conversions - from Self to C/C++

Conversion	Second part	Self type	C/C++ type	Errors	Notes
bool		boolean	int (0 or 1)	badTypeError	
char		smallInt	char	badTypeError overflowError	over- 1
signed_char		smallInt	signed char	badTypeError overflowError	over- flowError
unsigned_char		smallInt	unsigned char	badSignError bad- TypeError overflow- Error	bad- overflow-
short		smallInt	short	badTypeError overflowError	over- flowError
signed_short		smallInt	signed short	badTypeError overflowError	over- flowError
unsigned_short		smallInt	unsigned short	badSignError bad- TypeError overflow- Error	bad- overflow-
int		smallInt	int	badTypeError	
signed_int		smallInt	signed int	badTypeError	
unsigned_int		smallInt	unsigned int	badSignError bad- TypeError	bad-
long		smallInt	long	badTypeError	
signed_long		smallInt	signed long	badTypeError	
unsigned_long		smallInt	unsigned long	badSignError	
smi		smallInt	smi	badTypeError	2
unsigned_smi		smallInt	smi	badSignError bad- TypeError	bad- 2

Conversion	Second part	Self type	C/C++ type	Errors	Notes
float		float	float	badTypeError	3
double		float	double	badTypeError	3
long_double		float	long double	badTypeError	3
bv	ptr_type	byte vector	ptr_type	badTypeError	4
bv_len	ptr_type	byte vector	ptr_type, int	badSizeError TypeError	bad- 4, 5
bv_null	ptr_type	byte vector/0	ptr_type	badTypeError	4, 6
bv_len_null	ptr_type	byte vector/0	ptr_type, int	badSizeError TypeError	bad- 4, 5, 6
cbv	ptr_type	byte vector	ptr_type	badTypeError	7
cbv_len	ptr_type	byte vector	ptr_type, int	badSizeError TypeError	bad- 7
cbv_null	ptr_type	byte vector/0	ptr_type	badTypeError	7
cbv_len_null	ptr_type	byte vector/0	ptr_type, int	badSizeError TypeError	bad- 7
string		byte vector	char *	badTypeError nullCharError	8
string_len		byte vector	char *, int	badTypeError nullCharError	5, 8
string_null		byte vector/0	char *	badTypeError nullCharError	6, 8
string_len_null		byte vector/0	char *, int	badTypeError nullCharError	5, 6, 8
proxy	(ptr_type, type_seal)	proxy	ptr_type, != NULL	badTypeError TypeSealError, deadProxyError, nullPointerError	bad- 9
proxy_null	(ptr_type, type_seal)	proxy	ptr_type	badTypeError TypeSealError deadProxyError	bad- 9
any_oop		any object	oop		10
oop	oop subtype	corresponding object	oop (subtype)	badTypeError	11
any	C/C++ type	int/float/proxy/byte vector, int	int/float/ptr/ptr	badIndexError badTypeError ProxyError	dead- 12

Notes

1. The C type `char` has a system dependent range. Either 0..255 or -128..127.
2. The type `smi` is used internally in the virtual machine (a 30 bit integer).
3. Precision may be lost in the conversion.
4. The second part of the conversion is a C pointer type. The address of the first byte in the byte vector, cast to this pointer type, is passed to the foreign routine. It is the responsibility of the foreign routine not to go past the end of the byte vector. The foreign routine should not retain pointers into the byte vector after the call has terminated. Note: canonical strings can not be passed through a `bv` conversion (`badTypeError` will result). This is to ensure that they are not accidentally modified by a foreign function.
5. This conversion passes two values to the foreign routine: a pointer to the first byte in the byte vector, and an integer which is the length of the byte vector divided by `sizeof(*ptr_type)`. If the size of the byte vector is not a multiple of `sizeof(*ptr_type)`, `badSizeError` results.

6. In addition to accepting a byte vector, this conversion accepts the integer 0, in which case a `NULL` pointer is passed to the foreign routine.
7. The `cbv` conversions are like the `bv` conversions except that canonical strings are allowed as actual arguments. A `cbv` conversion should only be used if it is guaranteed that the foreign routine does not modify the bytes it gets a pointer to.
8. All the string conversions take an incoming byte vector, copy the bytes part, add a trailing null char, and pass a pointer to this copy to the foreign routine. After the call has terminated, the copy is discarded. If the byte vector contains a null char, `nullCharError` results.
9. The `type_seal` is an `int` or `char *` expression that is tested against the type seal value in the proxy. If the two are different, `badTypeSealError` results. The special value `ANY_SEAL` will match the type seal in any proxy. Note that the `proxy` conversion will fail with `nullPointerError` if the proxy object it is given encapsulates a `NULL` pointer.
10. The `any_oop` conversion is an escape: it passes the `Self` object unchanged to the foreign routine.
11. The `oop` conversion is mainly intended for internal use. The second argument is the name of an oop subtype. After checking that the incoming argument points to an instance of the subtype, the pointer is cast to the subtype.
12. The `any` conversion is different from all other conversions in that it expects two incoming `Self` objects. The actions of the conversion depends on the type of the first object in the following way. If the first object is an integer, the second argument must also be an integer; the two integers are converted to `C int`'s, the second is shifted 16 bits to the left and they are or'ed together to produce the result. If the first object is a float, it is converted to a `C float` and the second object is ignored. If the first object is a proxy, the result is the pointer represented by the proxy, and the second argument is ignored. If the first object is a byte vector, the second object must be an integer which is interpreted as an index into the byte vector; the result is a pointer to the indexed byte.

8.14.15 Result conversions - from C/C++ to Self

A result conversion is given a C or C++ value of a certain type and performs these actions to produce a corresponding `Self` object:

- check that the C/C++ value can be converted to a `Self` object with no overflow or other error occurring. If not, report the error.
- do the conversion, i.e., construct the `Self` object corresponding to the given C/C++ value.

Table 8.8 lists all the available result conversions. Each row represents one conversion, with the first two columns designating the conversion pair. The third column lists the type of C or C++ value that the conversion pair accepts. The fourth column lists the type of `Self` object the conversion produces. The fifth column lists the kind of errors that can occur during the conversion. Finally, the sixth column contains references to numbered notes. The notes are found in the paragraphs following the table.

Table 8.8: Result conversions - from C/C++ to Self

Conversion	Second part	C/C++ type	Self type	Errors	Notes
void		void	smallInt (0)		
bool		int	boolean		
char		char	smallInt		
signed_char		signed char	smallInt		
unsigned_char		unsigned char	smallInt		
short		short	smallInt		
signed_short		signed short	smallInt		
unsigned_short		unsigned short	smallInt		
int		int	smallInt	overflowError	
signed_int		signed int	smallInt	overflowError	
unsigned_int		unsigned int	smallInt	overflowError	
long		long	smallInt	overflowError	
signed_long		signed long	smallInt	overflowError	
unsigned_long		unsigned long	smallInt	overflowError	
smi		smi	smallInt	overflowError	
int_or_errno	n	int	int	a UNIX error	1
float		float	float		2
double		double	float		2
long_double		long double	float		2
string		char *	byte vector	nullPointerException	3
proxy	(ptr_type, type_seal)	ptr_type	proxy	nullPointerException	3, 4, 8
proxy_null	(ptr_type, type_seal)	ptr_type	proxy		4, 8
proxy_or_errno	(ptr_type, type_seal, n)	ptr_type	proxy	a UNIX error	4, 5, 8
fct_proxy	(ptr_type, type_seal, arg_count)	ptr_type	fctProxy	nullPointerException	3, 6, 8
fct_proxy_null	(ptr_type, type_seal, arg_count)	ptr_type	fctProxy		6, 8
oop		oop	corresponding object		7, 8

Notes

1. This conversion returns an integer value, unless the integer has the value n (the second part of the conversion; often -1). If the integer is n, the conversion interprets the return value as a UNIX error indicator. It then constructs a string describing the error (by looking at `errno`) and invokes the “IfFail block” with this string.
2. Precision may be lost.
3. This conversion fails with `nullPointerException` if attempting to convert a NULL pointer.
4. The `ptr_type` is the C/C++ type of the pointer. The `type_seal` is an expression of type `int` or `char *`. The conversion constructs a new proxy object, stores the C/C++ pointer in it and sets its type seal to be the value of `type_seal`.

5. If the pointer is `n` (often `n` is `NULL`), the conversion fails with a UNIX error, similar to the way `int_or_errno` may fail.
6. The `fct_proxy`, `fct_proxy_null` and `fct_proxy_or_errno` conversions are similar to the corresponding proxy conversions. The difference is that they produce a `fctProxy` object rather than a proxy object. Also, their second part is a triple rather than a pair. The extra component specifies how many arguments the function takes, if called. The special keyword `unknownNoOfArgs` or any nonnegative integer expression can be used here.
7. This conversion is an escape: it passes the C value unchanged to Self. It is an error to use it if the C value is not an `oop`.
8. The proxy (`fctProxy`) object that is returned by these conversions is *not* being created by the glue code. Rather a proxy (`fctProxy`) must be passed down from the Self level. This proxy (`fctProxy`), a *result proxy*, will then be side effected by the glue: the value that the foreign function returns will be stored in the result proxy together with the requested type seal. It is required that the result proxy is dead when passed down (else a `liveProxyError` results). After being side-effected and returned, the result proxy is live. The result proxy is the last argument of the function that the glue macro expands to.

8.14.16 A complete application using foreign functions

This section gives a description of a complete application which uses foreign functions. The aim is to present a realistic and complete example of how foreign functions may be used. The complete source for the example is found in the directory `objects/applications/serverDemo` in the Self distribution.

The example used is an application that allows Self expressions to be easily evaluated by non- Self processes. Having this, it then becomes possible to start Self processes from a UNIX prompt (shell) or to specify pipe lines in which some of the processes are Self processes. For example in

```
proto% cat someFile | tokenize | sort -r | capitalize | tee lst
```

it may be the case that the filters `tokenize` and `capitalize` perform most of their work in Self. Likewise, the command

```
proto% mail
```

may invoke some fancy mail reader written in Self rather than the standard UNIX mail reader.

To see how the above can be accomplished, please refer to Fig. 8.3 below. The left side of the figure shows the external view of a typical UNIX process. It has two files: `stdin` and `stdout` (for simplicity we ignore `stderr`). `Stdin` is often connected to the keyboard so that characters typed here can be read from the file `stdin`. Likewise, `stdout` is typically connected to the console so that the process can display output by writing it to the file `stdout`. `Stdin` and `stdout` can also be connected to “regular” files, if the process was started with redirection. The right side of Fig. 8.3 shows a two stage pipe line. Here `stdout` of the first process is connected to `stdin` of the second process.

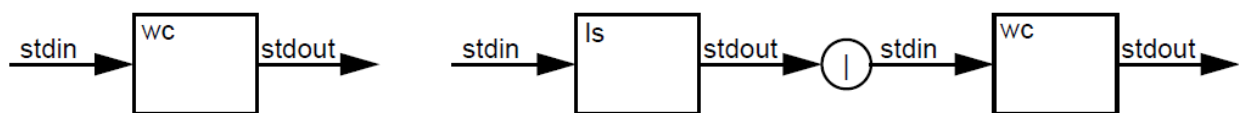


Fig. 8.3: A single UNIX process and an pipe line.

Fig. 8.3 illustrates a simple trick

that in many situations allows Self processes to behave as if they are full-fledged UNIX processes. A Self process is represented by a “real” UNIX process which transparently communicates with the Self process over a pair of connected sockets. The communication is bidirectional: input to the UNIX process is relayed to the Self process over

the socket connection, and output produced by the Self process is sent over the same socket connection to the UNIX process which relays it to stdout. The right part of Fig. 8.3 shows how the UNIX/Self process pair can fit seamlessly into a pipe line.

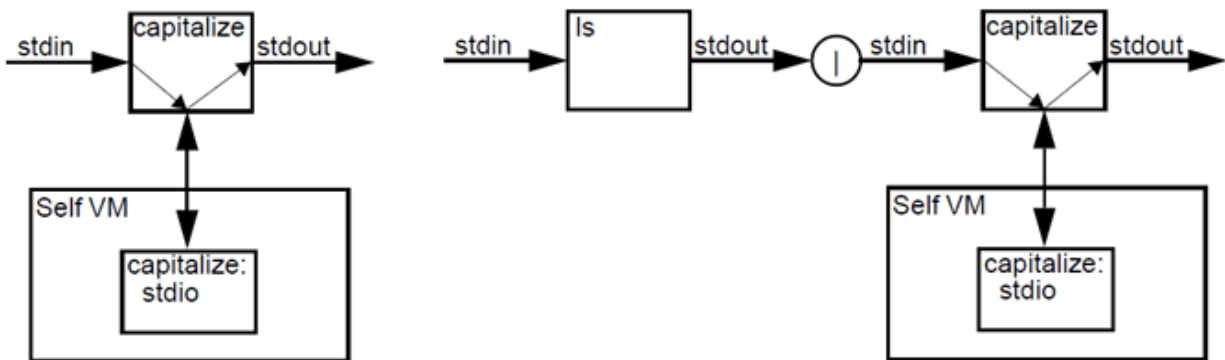


Fig. 8.4: A Self process and how it fits into a pipe line.

Source code that facilitates setting up such UNIX/Self process pairs is included in the Self distribution. The source consists of two parts: one being a Self program (called *server*), the other being a C++ program (called *toself*). When the server is started, it creates a socket, binds a name to it and then listens for connections on it. *toself* establishes connections to the server program. The first line that is transmitted when a connection has been set up goes from *toself* to the server. The line contains a Self expression. Upon receiving it, the server forks a new process to evaluate the expression in the context of the lobby augmented with a slot, *stdio*, that contains a `unixFile`-like object that represents the socket connection. When the forked process terminates, the socket connection is shut down. The *toself* UNIX process then terminates.

The Self expression that forms the Self process is specified on the command line when *toself* is started. For example, if the server has been started, the following can be typed at the UNIX prompt:

```
proto% toself stdio writeLine: 5 factorial printString
120

proto% echo something | toself capitalize: stdio
SOMETHING

proto% toself capitalize: stdio
Write some text that goes to stdin of the toself program
WRITE SOME TEXT THAT GOES TO STDIN OF THE TOSelf PROGRAM
More text
MORE TEXT
^D

proto%
```

If you want to try out these examples, locate the files `server.self`, `socks.so` and `toself`. The path name of the file `socks.so` is hardwired in the file `server.self` so please make sure that it has been set correctly for your system. Then file in the world and type `[server start] fork` at the Self prompt. Now you can go back to the UNIX prompt and try out the examples shown above.

8.14.17 Outline of `toself`

`toself` is a small C++ program found in the file `toself.c`. It operates in the three phases outlined above:

1. Try to connect to a well-known port number on a given machine (the function `establishConnection` does this).
2. Send the command line arguments over the connection established in 1 (the `safeWrite` call in `main` does this).
3. While there is more input and the Self process has not shut down the socket connection, relay from `stdin` to the socket connection and from the socket connection to `stdout` (the function `relay` does this).

8.14.18 Outline of server

The server is a Self program. It is found in the file `server.self`. When the server is started, the following happens:

1. Create a socket, bind a name to it and start listening.
2. **Loop: accept a connection and fork a new process (both step 1 and 2 are performed by the method `server start`).** The
 - (a) Reads a line from the connection.
 - (b) Sets up a context with a slot `stdio` referring to the connection.
 - (c) Evaluates the line read in step (a) in this context.
 - (d) Closes the connection.

8.14.19 Foreign functions and glue needed to implement server

The server program needs to do a number of UNIX calls to create sockets and bind names to them etc. The calls needed are `socket`, `bind`, `listen`, `accept` and `shutdown`. The first three of these are only called in a fixed sequence, so to make things easier, a small C++ function `socket_bind_listen`, that bundles them up in the right sequence, has been written. The `accept` function is more general than what is needed for this application, so a wrapper function, `simple_accept`, has been written. The result is that the server needs to call only three foreign functions: `socket_bind_listen`, `simple_accept` and `shutdown`. Glue for these three functions and the source for the first two is found in the file `socks.c`. This file is compiled and linked using the `Makefile`. The result is a shared object file, `socks.so`.

8.14.20 Use of foreign functions in `server.self`

The server program is implemented using `foreignFct` objects. There is only a few lines of code directly involved in setting this up. First the `foreignFct` prototype is cloned to obtain a “local prototype”, called `socksFct`, which contains the path for the `socks.so` file. `socksFct` is then cloned each time a `foreignFct` object for a function defined in `socks.so` is needed. For example, in `traits socket`, the following method is found:

```
copyPort: portNumber = ( "Create a socket, do bind, then listen."  
  | sbl = socksFct copyName: 'socket_bind_listen_glue'. |  
  sbl value: portNumber With: deadCopy.  
  ).
```

This method copies a `socket` object and returns the copy. The local slot `sbl` is initialized to a `foreignFct` object. The body of the method simply sends `value:With:` to the `foreignFct` object. The first argument is the port number to request for the socket, the second argument is a `deadCopy` of `self` (socket objects are proxies and `socket_bind_listen` returns a proxy, so it must be passed a `dead proxy` to revive and store the result in; see section *Proxy and fctProxy objects*).

There are only three uses of `foreignFct` objects in the server and in all three cases, the `foreignFct` object is encapsulated in a method as illustrated above.

In general the design of `foreignFct` objects has been aimed at making the use of them light weight. When cloning them, it is only necessary to specify the minimal information: the name of the foreign function. They can be encapsulated in a method thus localizing the impact of redesigns. The complications of dynamic loading and linking are handled automatically, as is the recovery of dead `fctProxies`.

REFERENCES

- [**APS93**] Ole Agesen, Jens Palsberg and Michael I. Schwartzbach. Type Inference of SELF. In ECOOP '93 Conference Proceedings, Kaiserslautern, Germany, July 1993. Published as Springer-Verlag LNCS 707, 1993.
- [**Age94a**] Ole Agesen. Mango: A Parser Generator for SELF. Sun Microsystems Labs TR SMLI TR-94-27, 1994.
- [**Age94b**] Ole Agesen. Constraint Based Type Inference and Parametric Polymorphism. In Proc. International Static Analysis Symposium, Sep 28-30, 1994.
- [**CU89**] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, OR, June, 1989. Published as SIGPLAN Notices 24(7), July, 1989.
- [**CU90**] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June, 1990. Also published in *Lisp and Symbolic Computation* 4(3), June, 1991.
- [**CU91**] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In OOPSLA '91 Conference Proceedings, Phoenix, AZ, October, 1991. Published as SIGPLAN Notices 26(11), November, 1991.
- [**CUC91**] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF. In *Lisp and Symbolic Computation* 4(3), June, 1991.
- [**CUL89**] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In OOPSLA '89 Conference Proceedings, New Orleans, LA, October, 1989. Published as SIGPLAN Notices 24(10), October, 1989. Also published in *Lisp and Symbolic Computation* 4(3), June, 1991.
- [**Cha92**] Craig Chambers. The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. Ph. D. dissertation, Computer Science Department, Stanford University, March 1992.
- [**CU93**] Bay-Wei Chang and David Ungar. Animation: From Cartoons to the User Interface. In UIST '93 Conference Proceedings, 1993.
- [**DS84**] L Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, Salt Lake City, UT, 1984.
- [**GR83**] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.
- [**HCU91**] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In ECOOP '91 Conference Proceedings, Geneva, Switzerland, July, 1991. Published as Springer-Verlag LNCS 512, 1991.

- [HCU92]** Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, June 1992. Published as SIGPLAN Notices 27(7), July, 1992.
- [Hoe94]** Urs Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Stanford University, August 1994.
- [HU94]** Urs Hölzle and David Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In Proceedings of OOPSLA '94, October 1994.
- [Lee88]** Elgin Lee. Object Storage and Inheritance for SELF. Engineer's thesis, Stanford University, 1988.
- [Ung84]** David Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April, 1984. Published as SIGPLAN Notices 19(5), May, 1984 and Software Engineering Notes 9(3), May, 1984.
- [Ung86]** David Ungar. The Design and Evaluation of a High Performance Smalltalk System. MIT Press, Cambridge, MA, 1987.
- [UCC91]** David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs without Classes. In Lisp and Symbolic Computation 4(3), June, 1991.
- [US87]** David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In OOPSLA '87 Conference Proceedings, Orlando, FL, 1987. Published as SIGPLAN Notices 22(12), December, 1987. Also published in Lisp and Symbolic Computation 4(3), June, 1991, and as Sun Microsystems Labs TR SMLI 94-0320.

10.1 Glossary

- A *slot* is a name-value pair. The value of a slot is often called its *contents*.
- An *object* is composed of a (possibly empty) set of slots and, optionally, a series of expressions called *code*. The Self implementation provides objects with indexable slots (vectors) via a set of primitives.
- A *data object* is an object without code.
- A *data slot* is a slot holding a data object. An *assignment slot* is a slot containing the *assignment primitive*. An *assignable data slot* is a data slot for which there is a corresponding assignment slot whose name consists of the data slot's name followed by a colon. When an assignment slot is evaluated its argument is stored in the corresponding data slot.
- An *ordinary method* (or simply *method*) is an object with code and is stored as the contents of a slot. The method's name (also called its *selector*) is the name of the slot in which it is stored.
- A *block* is an object representing a lexically-scoped closure (similar to a Smalltalk block).
- A *block method* is the method that is executed when a block is evaluated by sending it `value`, `value:`, `value:With:`, etc. A block method is a special kind of method that is evaluated within the scope of its method and any lexically enclosing blocks.
- An `activation` object records the state of an executing method or block method. It is a clone of the method prototype used to store the method's arguments and local slots during execution. There are two kinds of activation objects: `ordinary method activation` objects (or simply `method activation` objects) and `block method activation` objects.
- A `non-lifo block` is a block that is evaluated after the activation of its lexically enclosing block or method has returned. This results in an error in the current implementation.
- A `non-local return` is a return from a method activation resulting from performing a return (i.e., evaluating an expression preceded by the '^' operator) from within a lexically enclosed block. A non-local return forces returns from all activations between the method activation and the activation of the block performing the return.
- The `method holder` of a method is the object containing the slot holding that method.
- The `sending method holder` of a message is the method holder of the method that sent it.
- A `message` is a request to an object to perform some operation. The object to which the request is sent is called the `receiver`. A `message send` is the action of sending a message to a receiver.
- A *primitive send* is a message handled by invoking a *primitive*, a predefined function provided by the Self implementation.

- Messages that do not have an explicit receiver are known as *implicit-receiver messages*. The receiver is bound to `self`.
- A *unary message* is a message consisting of a single identifier sent to a receiver. A *binary message* is a message consisting of an operator and a single argument sent to a receiver. A *keyword message* is a message consisting of one or more identifiers with trailing colons, each followed by an argument, sent to a receiver.
- *Unary*, *binary*, and *keyword slots* are slots with selectors that match unary, binary, and keyword messages, respectively.
- An *argument slot* is a slot in a method filled in with a value when the method is invoked.
- *Message lookup* is the process by which objects determine how to respond to a message (which slot to evaluate), by searching objects for slots matching the message.
- *Inheritance* is the mechanism by which message lookup searches objects for slots when the receiver's slots are exhausted. An object's *parent slots* contain objects that it inherits from.
- *Dynamic inheritance* is the modification of object behavior by setting an assignable parent slot.
- A *resend* allows a method to invoke the method that the first method (the one that invokes the resend) is overriding. A *directed resend* constrains the lookup to search a single parent slot.
- *Cloning* is the primitive operation returning an exact shallow copy (a *clone*) of an object, i.e. a new object containing exactly the same slots and code as the original object.
- A *prototype* is an object that is used as a template from which new objects are cloned.
- A *traits object* is a parent object containing shared behavior, playing a role somewhat similar to a class in a class-based system. Any Self implementation is required to provide traits objects for integers, floats, strings, and blocks (i.e. one object which is the parent of all integers, another object for floats, etc.).
- The *root context* is the object that provides the context (i.e., set of bindings) in which slot initializers are evaluated. This object is known as the *lobby*. During slot initialization, `self` is bound to the lobby. The lobby is also the sending method holder for any sends in the initializing expression.
- *nil* is the object used to initialize slots without explicit initializers. It is intended to indicate “not a useful object.” This object is provided by the Self implementation.

10.2 Lexical overview

small-letter	→	'a' 'b' ... 'z'
cap-letter	→	'A' 'B' ... 'Z'
letter	→	small-letter cap-letter
identifier	→	(small-letter '_') { letter digit '_' }
small-keyword	→	identifier ':'
cap-keyword	→	cap-letter { letter digit '_' } ':'
argument-name	→	':' identifier
op-char	→	'!' '@' '#' '\$' '%' '^' '&' '*' '-' '+' '=' '~' '/' '?' '<' '>' ';' ':' ' ' '"' \'
operator	→	op-char { op-char }
number	→	['-'] (integer real)
integer	→	[base] general-digit { general-digit }
real	→	fixed-point float
fixed-point	→	decimal '.' decimal
float	→	decimal ['.' decimal] ('e' 'E') ['+' '-'] decimal
general-digit	→	digit letter
decimal	→	digit { digit }
base	→	decimal ('r' 'R')
string	→	"" { normal-char escape-char } ""
normal-char	→	any character except '\ and ""
escape-char	→	'\t' '\b' '\n' '\f' '\r' '\v' '\a' '\0' '\\' '\"' '\?' numeric-escape
numeric-escape	→	'\x' general-digit general-digit ('\d' '\o') digit digit digit
comment	→	"" { comment-char } ""
comment-char	→	any character but ""

10.3 Syntax overview

expression	→	constant unary-message binary-message keyword-message '(' expression ')'
constant	→	self number string object
unary-message	→	receiver unary-send resend '.' unary-send
unary-send	→	identifier
binary-message	→	receiver binary-send resend '.' binary-send
binary-send	→	operator expression
keyword-message	→	receiver keyword-send resend '.' keyword-send
keyword-send	→	small-keyword expression { cap-keyword expression }
receiver	→	[expression]
resend	→	resend identifier
object	→	regular-object block
regular-object	→	'(['!' ['{' '}' '=' string] slot-list '!'] [code])'
block	→	'[['!' slot-list '!'] [code] '
slot-list	→	{ unannotated-slot-list annotated-slot-list }
annotated-slot-list	→	'{ ' string slot-list '
unannotated-slot-list	→	{ slot '.' } slot ['.']
code	→	{ expression '.' } ['^'] expression ['.']
slot	→	arg-slot data-slot binary-slot keyword-slot
arg-slot	→	argument-name
data-slot	→	slot-name slot-name '<' expression slot-name '=' expression
unary-slot	→	slot-name '=' regular-object
binary-slot	→	operator '=' regular-object operator [identifier] '=' regular-object
keyword-slot	→	small-keyword { cap-keyword } '=' regular-object small-keyword identifier { cap-keyword identifier } '=' regular-object
slot-name	→	identifier parent-name
parent-name	→	identifier '*'

10.4 Built-in types

There are a small number of built-in types that are directly supported through primitives and syntax:

Integers and *floats* are provided with primitives for performing arithmetic operations, comparisons etc.

Strings have a *byte vector* part for storing the characters. Special string primitives are provided.

Blocks are objects which combine code with an environment link. Used for control structures, they are described in section langref-blocks.

In addition, there are a number of VM-supported types described in the sections on the Self World and the VM reference manual, such as *mirrors*, *processes*, *vectors*, *proxies* and *profilers*.

10.5 Useful Selectors

This is a list of selectors which Selfers should find useful as a starting point.

10.5.1 Copying

clone	shallow copy (for use within an object; clients should use copy)
copy	copy the receiver, possibly with embedded copies or initialization

10.5.2 Comparing

Equality

=	equal
!=	not equal
hash	hash value
==	identical (the same object; this is reflective and should be avoided)
!==	not identical

Ordered

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
compare: IfLess: Equal: Greater:	three way comparison
compare: IfLess: Equal: Greater: Incomparable:	three way comparison with failure

10.5.3 Numeric operations

+	add
-	subtract
*	multiply
/	divide
/=	divide exactly (returns float)
/~	divide and round to integer (tends to round up)
/+	divide and round up to integer
/-%	divide and round down to integer modulus
absoluteValue	absolute value
inverse	multiplicative inverse
negate	additive inverse
ceil	round towards positive infinity
floor	round towards negative infinity
truncate	truncate towards zero
round	round
asFloat	coerce to float
asInteger	coerce to integer
double	multiply by two
quadruple	multiply by four
half	divide by two
quarter	divide by four

Continued on next page

Table 10.1 – continued from previous page

min:	minimum of receiver and argument
max:	maximum of receiver and argument
mean:	mean of receiver and argument
pred	predecessor
predecessor	predecessor
succ	successor
successor	successor
power:	raise receiver to integer power
log:	logarithm of argument base receiver, rounded down to integer
square	square
squareRoot	square root
factorial	factorial
fibonacci	fibonacci
sign	signum (-1, 0, 1)
even	true if receiver is even
odd	true if receiver is odd

10.5.4 Bitwise operations (integers)

&&	and
	or
^^	xor
complement	bitwise complement
<<	logical left shift
>>	logical right shift
<+	arithmetic left shift
+>	arithmetic right shift

10.5.5 Logical operations (booleans)

&&	and
	or
^^	xor
not	logical complement

10.5.6 Constructing

@	point construction (receiver and argument are integers)
#	rectangle construction (receiver and argument are points)
##	rectangle construction (receiver is a point, argument is an extent)
&	collection construction (result can be converted into collection)
,	concatenation

10.5.7 Printing

print	print object on stdout
println	print object on stdout with trailing newline
printString	return a string label
printStringDepth:	return a string label with depth limitation request
printStringSize:	return a string label with number of characters limitation request
printStringSize: Depth:	return a string label with depth and size limitation request

10.5.8 Control

Block evaluation

value[: {With: }]	evaluate a block, passing arguments
-------------------	-------------------------------------

Selection

ifTrue:	evaluate argument if receiver is true
ifFalse:	evaluate argument if receiver is false
ifTrue: False:	evaluate first arg if true, second arg if false
ifFalse: True:	evaluate first arg if false, second arg if true

Local exiting

exit	exit block and return nil if block's argument is evaluated
exitValue	exit block and return a value if block's argument is evaluated

Basic looping

loop	repeat the block forever
loopExit	repeat the block until argument is evaluated; then exit and return nil
loopExitValue	repeat the block until argument is evaluated; then exit and return a value

Pre-test looping

whileTrue	repeat the receiver until it evaluates to true
whileFalse	repeat the receiver until it evaluates to false
whileTrue:	repeat the receiver and argument until receiver evaluates to true
whileFalse:	repeat the receiver and argument until receiver evaluates to false

Post-test looping

untilTrue:	repeat the receiver and argument until argument evaluates to true
untilFalse:	repeat the receiver and argument until argument evaluates to false

Iterators

do:	iterate, passing each element to the argument block
to: By: Do:	iterate, with stepping
to: Do:	iterate forward
upTo: By: Do:	iterate forward, without last element, with stepping
upTo: Do:	iterate forward, without last element
downTo: By: Do:	reverse iterate, with stepping
downTo: Do:	reverse iterate

10.5.9 Collections

Sizing

isEmpty	test if collection is empty
size	return number of elements in collection

Adding

add:	add argument element to collection receiver
addAll:	add all elements of argument to receiver
at: Put:	add key-value pair
at: Put: IfAbsent:	add key-value pair, evaluating block if key is absent
addFirst:	add element to head of list
addLast:	add element to tail of list
copyAddAll:	return a copy containing the elements of both receiver and argument
copyContaining:	return a copy containing only the elements of the argument

Removing

remove:	remove the given element
remove: IfAbsent:	remove the given element, evaluating block if absent
removeAll	remove all elements
removeFirst	remove first element from list
removeLast	remove last element from list
removeAllOccurrences:	remove all occurrences of this element from list
removeKey:	remove element at the given key
removeKey: IfAbsent:	remove element at the given key, evaluating block if absent
copyRemoveAll	return an empty copy

Accessing

first	return the first element
last	return the last element
includes:	test if element is member of the collection
occurrencesOf:	return number of occurrences of element in collection
findFirst: IfPresent: IfAbsent:	evaluate present block on first element found satisfying criteria, absent block if no such element
at:	return element at the given key
at: IfAbsent:	return element at the given key, evaluating block if absent
includesKey:	test if collection contains a given key

Iterating

do:	iterate, passing each element to argument block
doFirst: Middle: Last: IfEmpty:	iterate, with special behavior for first and last
doFirst: MiddleLast: IfEmpty:	iterate, with special behavior for first
doFirstLast: Middle: IfEmpty:	iterate, with special behavior for ends
doFirstMiddle: Last: IfEmpty:	iterate, with special behavior for last
reverseDo:	iterate backwards through list
with: Do:	co-iterate, passing corresponding elements to block

Reducing

max	return maximum element
mean	return mean of elements
min	return minimum element
sum	return sum of elements
product	return product of elements
reduceWith:	evaluate reduction block with elements
reduceWith: IfEmpty:	evaluate reduction block with elements, evaluating block if empty

Transforming

asByteVector	return a byte vector with same elements
asString	return a string with same elements
asVector	return a vector with same elements
asList	return a list with the same elements
filterBy: Into:	add elements that satisfy filter block to a collection
mapBy:	add result of evaluating map block with each element to this collection
mapBy: Into:	add result of evaluating map block with each element to a collection

Sorting

sort	sort receiver in place
copySorted	copy sorted in ascending order
copyReverseSorted	copy sorted in descending order
copySortedBy:	copy sorted by custom sort criteria
sortedDo:	iterate in ascending order
reverseSortedDo:	iterate in descending order
sortedBy: Do:	iterate in order of custom sort criteria

Indexable-specific

firstKey	return the first key
lastKey	return the last key
loopFrom: Do:	circularly iterate, starting from element n
copyAddFirst:	return a copy of this collection with element added to beginning
copyAddLast:	return a copy of this collection with element added to end
copyFrom:	return a copy of this collection from element n
copyFrom: UpTo:	return a copy of this collection from element n up to element m
copyWithoutLast	return a copy of this collection without the last element
copySize:	copy with size n
copySize: FillingWith:	copy with size n, filling in any extra elements with second arg

10.5.10 Timing

realTime	elapsed real time to execute a block
cpuTime	CPU time to execute a block
userTime	CPU time in user process to execute a block
systemTime	CPU time in system kernel to execute a block
totalTime	system + user time to execute a block

10.5.11 Message Sending

Sending

Like Smalltalk `perform`; receiver is a string.

sendTo: {With: }	send receiver string as a message
sendTo: WithArguments:	indirect send with arguments in a vector
sendTo: DelegatingTo: {With: }	indirect delegated send
sendTo: DelegatingTo: WithArguments:	indirect delegated send with arg vector
resendTo: {With: }	indirect resend
resendTo: WithArguments:	indirect resend with arguments in a vector

Message object protocol

send	perform the send described by a message object
fork	start a new process; the new process performs the message
receiver:	set receiver
selector:	set selector
methodHolder:	set method holder
delegatee:	set delegatee of the message object
arguments:	set arguments (packaged in a vector)
receiver: Selector:	set receiver and selector
receiver: Selector: Arguments:	set receiver, selector, and arguments
receiver: Selector: Type: Delegatee: MethodHolder: Arguments:	set all components

10.5.12 Reflection (mirrors)

reflect:	returns a mirror on the argument
reflectee	returns the object the mirror receiver reflects
contentsAt:	returns a mirror on the contents of slot n
isAssignableAt:	tests if slot n is an assignable slot
isParentAt:	tests if slot n is a parent slot
isArgumentAt:	tests if slot n is an argument slot
parentPriorityAt:	returns the parent priority of slot n
slotAt:	returns a slot object representing slot n
contentsAt:	returns the contents of the slot named n
visibilityAt:	returns a visibility object representing visibility of slot n

10.5.13 System-wide Enumerations

Messages sent to the oddball object browse.

all[Limit:]	returns a vector of mirrors on all objects in the system (up to the limit)
referencesOf: [Limit:]	returns a vector of mirrors on all objects referring to arg (up to the limit)
referencesOfReflectee: [Limit:]	returns a vector of mirrors on all objects referring to argument's reflectee (up to the limit); allows one to find references to a method
childrenOf: [Limit:]	returns a vector of mirrors on all objects with a parent slot referring to the given object (up to the limit)
implementorsOf: [Limit:]	returns a vector of mirrors on objects with slots whose names match the given selector (up to the limit)
sendersOf: [Limit:]	returns a vector of mirrors on methods whose selectors match the given selector (up to the limit)

10.5.14 Debugging

halt	halt the current process
halt:	halt and print a message string
error:	halt, print an error message, and display the stack
warning:	beep, print a warning message, and continue

10.5.15 Virtual Machine-Generated

Errors

undefinedSelector: Type: Delegate: MethodHolder: Arguments:	lookup found no matching slot
ambiguousSelector: Type: Delegate: MethodHolder: Arguments:	lookup found more than one matching slot
missingParentSelector: Type: Delegate: MethodHolder: Arguments:	parent slot through which resend was delegated was not found
performTypeErrorSelector: Type: Delegate: MethodHolder: Arguments:	first argument to the <code>_Perform</code> primitive was not a canonical string
mismatchedArgumentCountSelector: Type: Delegate: MethodHolder: Arguments:	number of args supplied to <code>_Perform</code> primitive does not match selector
primitiveFailedError: Name:	the named primitive failed with given error string

Other system-triggered messages

postRead	slot to evaluate after reading a snapshot
----------	---

10.6 Every Menu Item in the Programming Environment

This table only covers the middle-button menus, the right-button (morph) menu is described elsewhere. It merges items from several menus: the background menu, the outliner whole-object menu, the outliner category menu, the outliner slot menu, the text editor menu, the debugger stack menu, the iterator object menus, and the changed module morph menu.

Table 10.2: Menu Items

Label	Function
Add Category	Adds a category to an object or category.
Add Slot	Adds a slot to an object or category.
Added or Changed Slots	On a module morph, enumerates slots added/changed since last save.
All Modules	Summons a hierarchical list of all modules from the changed modules morph.
All Slots	On a module morph, enumerates its slots.
Changed Modules	Summons a list of changed modules.
Children	Enumerate an object's children.
Clean Up	Clean up the screen: collapse outliners and stack them on the left of the window.
Clean Up Memory	Manually initiate a garbage collection. Can help when you know you have just freed up a bunch of space. Self also does this automatically.
Collapse All	Collapses all outliners, or all categories within an outliner or category.
Copy Down Children	Enumerate an object's copy-down children.
Copy Down Parent	Show an object's copy-down parent.
Copy	Copies slots, categories or text.

Continued on next page

Table 10.2 – continued from previous page

Label	Function
Core Sampler	Summons an object for manipulating morphs.
Create Button	For a slot, create a button to send the message to the object. The receiver may be set by carrying the button on top of the receiver and using the middle-button on the button. (The button is grabbed with either the car-pet-morph or with the grab right-menu item. Bug: buttons do not mani-fest their results.)
Cut	Copies text to the text buffer.
Do Selection	Evaluate the selected text, do not show the result.
Do it	Evaluate the text in the editor, do not show the result.
Edit	On a slot, open an editor to change its name, slot type, or contents.
Evaluator	Adds an evaluator window to an object outliner.
Expand All	Expand all subcategories.
Expatriate Slots	On the changed module morph; shows a list of slots not included in any module.
(Don't) Filter Frames	On a debugger stack, enable (or disable) filtering.
Find Slot	Searches an object and its ancestors for slot names matching a pattern.
Find Slot of :	For an assignable slot x, show all slots named x: in the object and its ancestors.
Flush	Discards cached state, e.g. the result of an enumeration.
Forget I was changed	On a module morph, removes it from the list of changed modules and clears out its record of added, changed & removed slots.
Get Module Object	On a module morph summons the object outliner for the module. Useful for editing its postFileIn method, or its revision.
Get Selection	Evaluate the selected text & show the result.
Get it	Evaluate the text in the editor, show the result.
Hide Annotation	Hides the object or slot annotation.
Hide Comment	Hides the object or slot comment.
Implementors	Searches for slots of a given name.
Implementors of :	For an assignable slot x, show all implementors of x:.
Load Morph From File	Reads in a file created with the right-menu item "Save Morph to File"
Make Creator	On a slot, set the creator annotation of its contents to be the slot.
Make Private	Change the style of the slot to show that it is intended to be private (not enforced).
Make Public	Change the style of the slot to show that it is intended to be public (not enforced). Adds a comment for posterity.
Make Undeclared	Change the style of the slot to show that no clear intention exists as to its visibility. (A Self exclusive!)
Methods Containing	Searches for all methods containing a string.
Move	Moves slots or categories.
New Shell	Summon a new shell object.
Open Factory Win-dow	Open a new window containing handy morphs (such as a radar-view) you can tear-off and drag to other Self windows.
Palette	Summons an object for obtaining morphs for building.
Paste	Pastes text from the buffer.
Quit	Leave job and ride boxcars.
Radar View	Summons an object for moving the current viewport around in space.
Read Module	On a module morph, rereads the source file.
References	Enumerate references to an object.
Removed Slots	On a module morph, lists removed slot paths.

Continued on next page

Table 10.2 – continued from previous page

Label	Function
Restore Window State	Restores the saved state of the screen.
Save snapshot	Saves an image of all objects in a snapshot file. Overwrites the snapshot file that was opened originally. Saves the previous version with a ".old" suffix.
Save snapshot as ...	Lets you set the file name and other parameters of the saved snapshot. For example, if you have a lot of memory, you can increase the code cache size.
Save Window State	Saves the state of the screen.
Send	For a method in a concrete object, send the message to the object.
Senders	Searches for methods sending a given message.
Senders of :	For an assignable slot x, show all senders of x:, i.e. methods that might assign to x.
Senders in family	Searches for methods sending a given message in the selected object, its ancestors, and its descendants.
Senders of : in family	For an assignable slot x, show all senders of x:, i.e. methods that might assign to x in the selected object, its ancestors, and its descendants.
Set Module	Sets the module of a slot or group of slots.
Shell	Summons an outliner on the shell. Used for evaluating expressions.
Show All Frame	On a debugger stack, disable filtering.
Show Annotation	Shows the object or slot annotation.
Show Comment	Shows the object or slot comment.
Show Morph	For morph object outliners, summons the morph that the object implements.
"Subclass" Me	Appears on the object menu. Automates several steps equivalent to sub-classing in Smalltalk: Creates a copy-down child of the selected object and makes a new parent object for the new child that inherits from the selected object's parents. It also sets some of the annotations for transport.
The box at the top.	Pins up the menu.
Toggle Spy	Toggles an X Window spying on the Virtual Machine. A nice source of reassurance.
Traits Family	Show an inheritance hierarchy textually. Only works on certain objects on alternate Thursdays.
Write Snapshot	Saves all the objects in the Self world to a (fairly large) file.

10.7 The system monitor

The Self system contains a system monitor to display information about the internal workings of the system such as memory management and compilation. It is invoked with `_Spy: true` (there are shortcuts in the shell, `spyOn` and `spyOff`). When it is active, the system monitor takes over a portion of your screen with a window that looks like this:

The indicators in the left part of the display correspond to various internal activities and events. On the very left are the CPU bars which show how much CPU is used in various parts of the system. The following table lists the individual indicators:

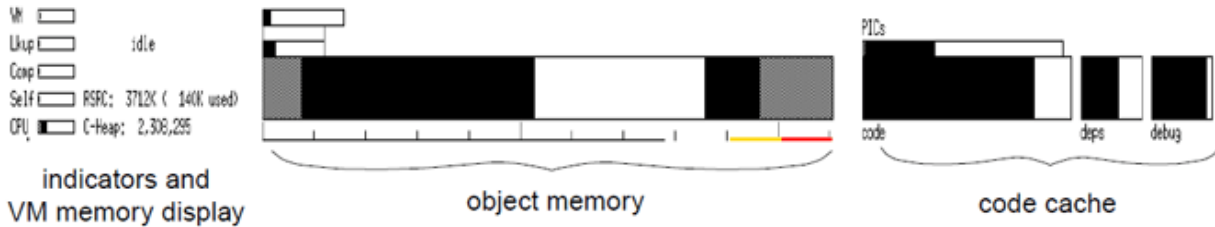


Table 10.3: The system monitor display: indicators

CPU	Bar	What It Means
VM		CPU time spent executing in the VM, i.e. for primitives, garbage collection etc.
Lkup		CPU time used by compile-time and run-time lookups.
Comp		CPU time spent by the Self compilers. The black part stands for time consumed by the non-inlining compiler (NIC), the gray part for the simple inlining compiler (SIC).
Self		CPU time spent executing compiled Self code. The black part stands for time consumed by unoptimized (NIC) code, the gray part for optimized (SIC) code.
CPU		This bar displays the percentage of the CPU that the Self process is getting (a completely filled bar equals 100% CPU utilization by Self). Black stands for user time, gray for system time.
Dot		Below the CPU bar is a small dot which moves whenever a process switch takes place.
Indicator	What It Means	
X-compiling Y	The X compiler (where X is either “nic” or “sic”) is compiling the method named Y into machine code.	
scavenge	The Self object memory is being scavenged. A scavenge is a fast, partial garbage collection (see [Ung84], [Ung86], [Lee88]).	
GC	The Self object memory is being fully garbage-collected.	
flushing	Self is flushing the code cache.	
compacting	Self is compacting the code cache.	
reclaiming	Self is reclaiming space in the code cache to make room for a new method.	
sec reclaim	Self is flushing some methods in the code cache because there is not enough room in one of the secondary caches (the caches holding the debugging and dependency information).	
ic flush	Self is flushing all inline caches.	
LRU sweep	Self is examining methods in the code cache to determine whether they have been used recently.	
page N N	page faults occurred during the last time interval (N is not displayed if N=1). The time interval currently is 1/25 of a second.	
read	Self is blocked reading from a “slow” device, e.g., the keyboard or mouse.	
write	Self is blocked writing to a “slow” device, e.g., the screen.	
disk in/out	Self is doing disk I/O.	
UNIX	Self is blocked in some UNIX system call other than read or write.	
idle	Self has nothing to do. (shows up only when using processes.)	

The middle part of the display contains some information on VM memory usage displayed in textual form, as described

below:

Table 10.4: VM memory status information

Name	Description
RSRC	Size and utilization of the resource area (an area of memory used for temporary storage by the compiler and by primitives).
C-Heap	Number of bytes allocated on the C heap by Self (excluding the memory and code spaces and the resource area).

The memory status portion of the system monitor consists of bars representing memory spaces and their utilization; all bars are drawn to scale relative to one another, their areas being proportional to the actual sizes of the memory spaces. The next table explains the details of this part of the system monitor's display.

Table 10.5: The system monitor display: memory status

Space	Description
object memory	The four (or more) bars represent (from top to bottom) eden, the two survivor spaces, and subsequent bars are segments of old space. The left and right parts of each bar represent the space used by "plain" objects and byte vectors, respectively. The above picture shows a situation in which about half of old space is filled with plain objects and about 25% is filled with byte vectors. A fraction of old space's used portions is currently paged out (gray areas). Below the old space is a ruler, marked in 1Mb intervals, showing the total allocated in old space (extending line at the left). To the right is a red bar representing how much of old space is reserved for use by the Virtual Machine, and a yellow bar representing the low space threshold (when crossed, the scheduler is notified and a garbage collection may take place).
code cache	These four bars represent the cache holding compiled methods with their associated debugging and dependency information. The bar labelled 'code' represents the cache containing the actual machine code for methods (including some headers and relocation information); it is divided into code generated by the primary (non-inlining) compiler, or NIC, and code generated by the secondary, smarter compiler (SIC). The cache represented by the bar labelled 'deps' contains dependency information for the compiled methods, and the cache represented by the bar labelled 'debug' contains the debugging information. The three-way split reduces the working set size of the code cache. The cache represented by the bar labelled 'PICs' contains polymorphic inline caches.
Color	Meaning
black	Allocated, residing in real memory.
gray	Allocated, paged out.
white	Unallocated memory.

10.8 Primitives

Primitives are Self methods implemented by the virtual machine. The first character of a primitive's selector is an underscore ('_'). You cannot define primitives yourself (unless you modify the Virtual Machine), nor can you define slots beginning with an underscore.

10.8.1 Primitive failures

Every primitive call can take an optional argument defining how errors should be handled for this call. To do this, the primitive is extended with an `IfFail:` argument. For example, `_AsObject` becomes `_AsObjectIfFail:`, and `_IntAdd:` becomes `_IntAdd:IfFail:`.

```
> 3 _IntAdd: 'a' IfFail: [ | :error. :name |
(name, ' failed with ', error, '.') printLine. 0 ]
_IntAdd: failed with badTypeError.
0      "The primitive returns the result of evaluating the failure block."
>
```

When a primitive fails, if the primitive call has an `IfFail:` part, the message `value:With:` is sent to the `IfFail:` argument, passing two strings: the name of the primitive and an error string indicating the reason for failure. If the failing primitive call *does not* have an `IfFail:` part, the message `primitive:FailedWith:` is sent to the receiver of the primitive call with the same two strings as arguments.

The result returned by the error handler becomes the result of the primitive operation (0 in our example); execution then continues normally. If you want the program to be aborted, you have to do this explicitly within the error handler, for example by calling the standard `error:` method defined in the default world.

The following table lists the error string prefixes passed by the VM to indicate the reason of the primitive failure. If the error string consists of more than the prefix it will reveal more details about the error.

Table 10.6: Primitive failures

Prefix	Description
<code>primitiveNotDefinedError</code>	Primitive not defined.
<code>primitiveFailedError</code>	General primitive failure (for example, an argument has an invalid value).
<code>badTypeError</code>	The receiver or an argument has the wrong type.
<code>badTypeSealError</code>	Proxy's type seal did not match expected type seal.
<code>divisionByZeroError</code>	Division by zero.
<code>overflowError</code>	Integer overflow. This can occur in integer arithmetic primitives or in UNIX (when the result is too large to be represented as an integer).
<code>badSignError</code>	Integer receiver or argument has wrong sign.
<code>alignmentError</code>	Bad word alignment in memory.
<code>badIndexError</code>	The vector index (e.g. in <code>_At:</code>) is out of bounds (too large or negative).
<code>badSizeError</code>	An invalid size of a vector was specified, e.g. attempting to clone a vector with a negative size (see <code>_Clone:Filler:</code> and <code>_CloneBytes:Filler:</code> below).
<code>reflectTypeError</code>	A mirror primitive was applied to the wrong kind of slot, e.g. <code>_MirrorParentGroupAt:</code> to a slot that isn't a parent slot.
<code>outOfMemoryError</code>	A primitive could not complete because its results would not fit in the existing space.
<code>stackOverflowError</code>	The stack overflowed during execution of the primitive or program.
<code>slotNameError</code>	Illegal slot name.
<code>argumentCountError</code>	Wrong number of arguments.
<code>unassignableSlotError</code>	This slot is not assignable.
<code>lonelyAssignmentSlotError</code>	Assignment slot must have a corresponding data slot.

Continued on next page

Table 10.6 – continued from previous page

Prefix	Description
<code>parallelTWINSError</code>	Can not invoke TWINs primitive (another process is already using it).
<code>noProcessError</code>	This process does not exist.
<code>noActivationError</code>	This method activation does not exist.
<code>noReceiverError</code>	This activation has no receiver.
<code>noParentSlot</code>	This activation has no lexical parent.
<code>noSenderSlot</code>	This activation has no sender slot.
<code>deadProxyError</code>	This proxy is dead and can not be used.
<code>liveProxyError</code>	This proxy is live and can not be used to hold a proxy result.
<code>wrongNoOfArgsError</code>	Wrong number of arguments was supplied with call of foreign function.
<code>nullPointerError</code>	Foreign function returned null pointer.
<code>nullCharError</code>	Can not pass byte vector containing null char to foreign function expecting a string.
<code>prematureEndOfInputError</code>	Premature end of input during parsing.
<code>noDynamicLinkerError</code>	Primitive depends on dynamic linker which is not available in this system.
<code>EPERM, ENOENT, ...</code>	These errors are returned by a UNIX primitive if a UNIX system call executed by the primitive fails. The UNIX error codes are defined in <code>/usr/include/sys/errno.h</code> ; see this file for details on the roughly 90 different UNIX error codes.

The `_ErrorMessage` primitive, sent to an error string returned by any primitive, returns a more descriptive version of the error message; this is especially useful for UNIX errors.

10.8.2 Available primitives

A complete list of primitives can be obtained by sending `primitiveList` to `primitives`. Documentation for a primitive (such as `_Clone`), can be obtained using `at:`, thus:

```
primitives at: '_Clone'
```

A list of primitive names matching a pattern can be obtained thus:

```
primitives match: '_Memory*'
```

Some points to note when browsing primitives:

- Since strings are special kinds of byte vectors, primitives taking byte vectors as arguments can usually take strings. The exception is that canonical strings cannot be passed to primitives that modify the object.
- Integer arithmetic primitives take integer receivers and arguments; floating-point arithmetic primitives take floating-point receivers and arguments.
- All comparison primitives return either true or false. Integer comparison primitives take integer receivers and arguments; floating-point comparison primitives take floating-point receivers and arguments.
- The receiver of a mirror primitive must be a mirror (unless otherwise noted)

Last updated 4 February 2014 for Self 4.5.0

The default Self source code tree comes with a number of extra objects and features which aren't included in the default Self World. These can in general be loaded by doing:

```
'path/to/file.self' runScript
```

or:

```
bootstrap load: 'file' From: 'path/to'
```

where the current working directory is the *objects* directory of the source code tree.

These extra features include:

11.1 The Original Self UI

Last updated 4 February 2014 for Self 4.5.0

Before Self used the current *morphic* user interface (also known as *UI2*) it used another, nameless, user interface framework which has been retrospectively dubbed *ui1*.

11.1.1 X11 with 256 Colors

UI1 only runs on 8 bit X11 systems. There are a number of suggested ways to get a 8 bit X11 window on modern systems detailed at [WineHQ](#).

On OS X systems, you will need to install XQuartz if you are running Mountain Lion (10.8) or later. You will need to set the color depth to 256 colors in the Preferences, then restart XQuartz.

11.1.2 VM with X11 support

The standard Linux VM already has X11 primitives. However the standard Mac VM doesn't, so that users can run Self on Macs without requiring X11 to be installed. You can either build a OS X VM which includes X11 support yourself as described [the section on building the VM](#) or download one from the Self fileserver.

11.1.3 Preparing a snapshot

The first step in using UI1 is to build a new World including it. Assuming you are in the *objects* directory in the Self source tree, and that the Self vm is in your path, do:

```
Self -f worldBuilder.self
```

and when it asks you:

```
Load UI1 (X11 only)? (y/N)
>
```

answer *y*. When the World has built, save it at the prompt by doing:

```
save
quitNoSave
```

This will save a snapshot of the world in the file `'Snapshot.snap'`.

11.1.4 Starting the UI

Once you have a 256 colour X11 desktop running, you can start your Self world and start using UI1 by:

```
preferences xDisplay: ':1' "Only if necessary"
ui demo
```