The SELF 4.1 Programmer’s Reference Manual

Ole Agesen
Lars Bak
Craig Chambers
Bay-Wei Chang
Urs Hölzle
John Maloney
Randall B. Smith
David Ungar
Mario Wolczko

Table of Contents

1 Introduction

1.1 Overview of the SELF System

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.

1.1.1 The system

Although SELF runs as a single UNIX[†] 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:

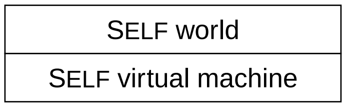


Figure 1 The SELF system

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

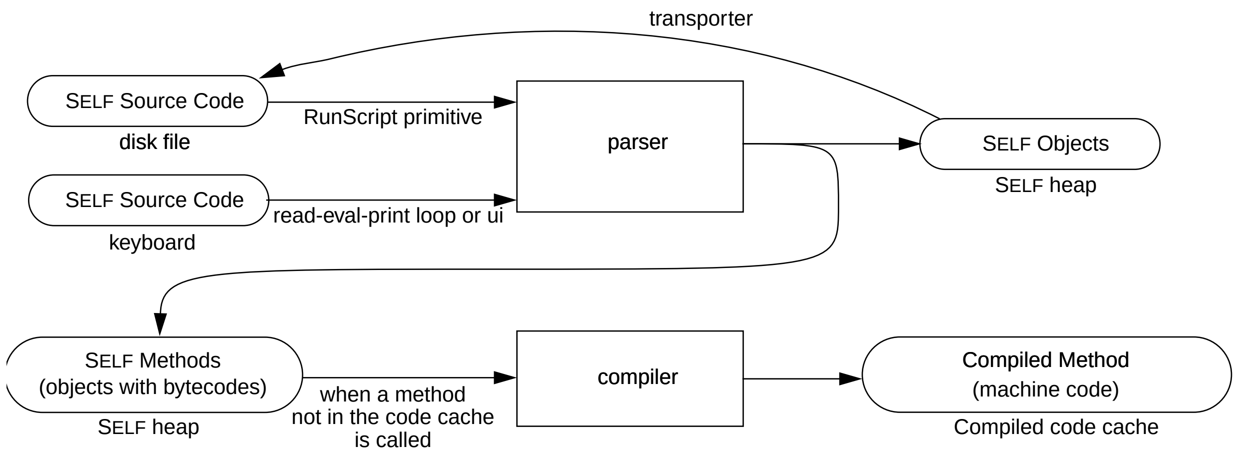


Figure 2 How SELF programs are compiled

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

1.1.2 The translation process

SELF programs are translated to machine code in a two-stage process (see Figure 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).

2 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 meta-symbols:

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
$\rightarrow$	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.

2.1 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* (§2.1.7) are SELF closures used to implement user-defined control structures. An object is composed of a (possibly empty) set of *slots* and, optionally, *code* (§2.1.5). A slot is a name-value pair; slots contain references to other objects. When a slot is found during a *message lookup* (§2.3.6) 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.

2.1.1 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 object.[†]

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

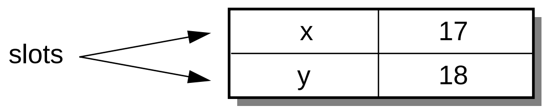
```
[ | slot1. slot2 | 'here is some code in a block' printLine ]
```

A *slot list* consists of a (possibly empty) sequence of *slot descriptors* (§2.2) separated by periods. A period at the end of the slot list is optional.[‡]

The code for an object is a sequence of *expressions* (§2.3) 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 (§2.1.8).

2.1.2 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.

2.1.3 The assignment primitive

A slot containing the assignment primitive is called an *assignment slot* (§2.2.2). When an assignment slot is evaluated, the argument to the message is stored in the corresponding *data slot* (§2.2) in the same object (the slot whose name is the assignment slot's name minus the trailing colon), and the *receiver* (§2.3) 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.)

[†] 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.

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

2.1.4 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.

2.1.5 Code

Code is a sequence of *expressions* (§2.3). 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.

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 4.

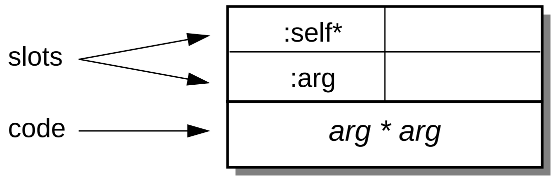
2.1.6 Methods

Ordinary methods (or simply “methods”) are methods that are not embedded in other code. A method can have *argument slots* (§2.2.3) and/or local slots. An ordinary method always has an implicit *parent* (§2.2.4) 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 )`:

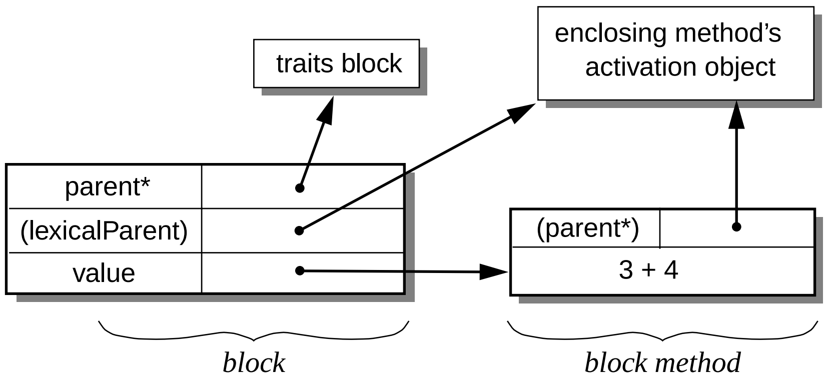


This method has an argument slot `arg` and returns the square of its argument.

2.1.7 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* (§2.3.4) 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 like:†



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.

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

2.1.8 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:

```
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.

2.1.9 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.

2.2 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 send (§2.3.3), and *parent slots* specify inheritance relationships (§2.3.8).

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

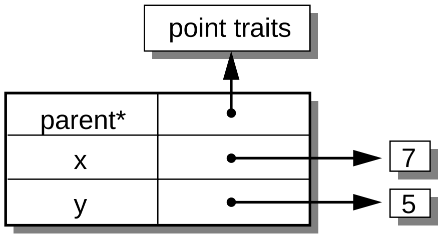
2.2.1 Read-only slots

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:



2.2.2 Read/write slots

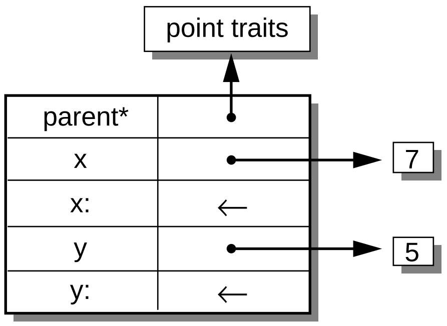
There is no separate assignment operation in SELF. Instead, assignments to data slots are message sends that invoke the assignment primitive. For example, a data slot `x` is assignable if and only if there is a slot in the same object with the same name appended with a colon (in this case, `x:`), containing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

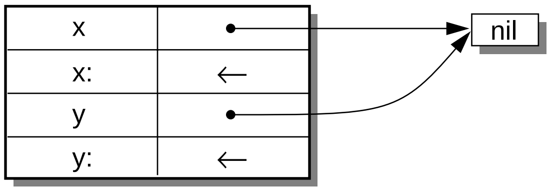


An identifier by itself specifies an assignable data slot initialized to *nil*.[‡] Thus, the slot declaration `x` is a shorthand notation for `x <- nil`.

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

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

producing:



2.2.3 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 slot.[*] 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:

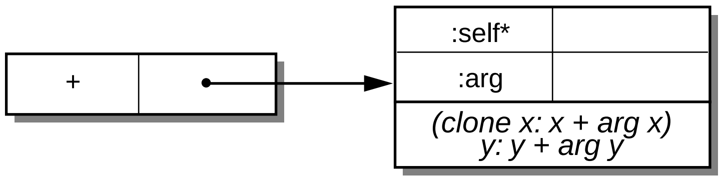
```
( |
            + = ( | :arg | (clone x: x + arg x) y: y + arg y ).
| )
```

[†] 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.

[‡] Nil is a predefined object provided by the implementation. It is intended to indicate “not a useful object.”

[*] 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 §2.1.7.

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* (§2.3.2), taking one argument and having a name that consists of operator symbols. Slots like `x` or `y` in a point object are *unary slots* (§2.3.1), which take no arguments and have simple identifiers for names. In addition, there are *keyword slots* (§2.3.3), 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.

2.2.4 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 (§2.3.8), parent slots are exactly like normal unary slots; in particular, they may be assignable, allowing *dynamic inheritance*. Argument slots cannot be parent slots.

2.2.5 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).[†]

2.3 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.

Productions:[‡]

[†] 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.

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

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 *	[*receiver*] *operator expression*
keyword	≥ 1	lowest	right-to-left	[*receiver*] *small-keyword expression* { *cap-keyword expression* }

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

Parentheses can be used to explicitly specify order of evaluation.

2.3.1 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.

2.3.2 Binary messages

A *binary message* has a receiver and a single argument, separated by a binary operator.

Examples of binary messages:

```
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)
```

2.3.3 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* (§2.3.7).

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))
```

2.3.4 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 §2.1.4 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.

2.3.5 Resending messages

A *resend* allows an overridding 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 resend.[†]

2.3.6 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.

[†] General delegation for explicit receiver messages is supported through primitives in the implementation (see Appendix 5.B).

$\{s \in obj \mid pred(s)\}$	The set of all slots of object *obj* that satisfy predicate *pred*.
$\mid S \mid$	The cardinality of set *S*.

The message sending semantics are decomposed into the following functions:

send(*rec, sel, args*)	The message send function (§2.3.7).
lookup(*obj, rec, sel, V*)	The lookup algorithm (§2.3.8).
undirected_resend(...)	The undirected message resend function (§2.3.9).
directed_resend(...)	The directed message resend function (§2.3.9).
eval(*rec, M, args*)	The slot evaluation function as described informally throughout §2.1.

2.3.7 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 §2.1.6.

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 $M \leftarrow lookup(rec, sel, \emptyset)$                      “do the lookup”
    case
        $|M| = 0$:   error: message not understood
        $|M| = 1$:   $res \leftarrow eval(rec, M, args)$             “see §2.1”
        $|M| > 1$:   error: ambiguous message send
    end
end
return $res$
```

2.3.8 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 $obj \in V$
then $M \leftarrow \emptyset$                                              “cycle detection”
else $M \leftarrow \{s \in obj \mid s.name = sel\}$                            “try local slots”
     if $M = \emptyset$ then $M \leftarrow parent\_lookup(obj, sel, V)$ end     “try parent slots”
end
return $M$
```

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

```
$P \leftarrow \{s \in obj \mid s.isParent\}$                              “all parents”
$M \leftarrow \bigcup_{s \in P} lookup(s.contents, sel, V \cup \{obj\})$        “recursively search parents”
return $M$
```

2.3.9 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:

```
$M \leftarrow parent\_lookup(smh, sel, \emptyset)$                     “do the lookup”
case
    $|M| = 0$:    error: message not understood
    $|M| = 1$:    $res \leftarrow eval(rec, M, args)$                “see §2.1”
```

```
        | $M$ | > 1:    error: *ambiguous message send*
    **end**
    **return** *res*
```

2.3.10 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:

```
    $D \longleftarrow \{s \in smh \mid s.name = del\}$                 “find delegatee”
    if | $D$ | = 0   then error: missing delegatee                  “one or none”
    $M \longleftarrow lookup(smh.del, sel, \varnothing)$               “do the lookup”
    case
        | $M$ | = 0:    error: message not understood
        | $M$ | = 1:    $res \longleftarrow eval(rec, M, args)$        “see §2.1”
        | $M$ | > 1:    error: ambiguous message send
    end
    return res
```

2.4 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.

2.4.1 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:
  !@#$%^&*()_-+=|\~‘{}[]:;"’<>,.?/

2.4.2 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.

2.4.3 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:`

2.4.4 Arguments

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

Production:

arg-name → ‘:’ *identifier*

Example: `:name`

2.4.5 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: `+ - && || <-> % # @ ^`

2.4.6 Numbers

Integer literals are written as a sequence of digits, optionally prefixed with a minus sign and/or a base.[†] No whitespace is allowed between a minus sign and the digit sequence.[‡] 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.,

$nnnn\mathrm{E}ddd = nnnn \times 10^{ddd}$

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 ± infinity or zero, respectively.

Productions:

number → [‘-’] (*integer* | *real)*

integer → [*base*] *general-digit* {*general-digit*}

[†] Unlike Smalltalk, integer literals are limited in range to smallInts.

[‡] 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`).

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’)

Examples: `123 16r27fe 1272.34e+15 1e10`

2.4.7 Strings

String constants are enclosed in single quotes (‘`''`’). With the exception of single quotes and escape sequences introduced by a backslash (‘`\`’), all characters (including formatting characters like newline and carriage return) lying between the delimiting single quotes are included in the string.[†]

To allow single quotes to appear in a string and to allow non-printing control characters in a string to be indicated more visibly, SELF provides C-like escape sequences:

`\t`	tab	`\b`	backspace	`\n`	newline
`\f`	form feed	`\r`	carriage return	`\v`	vertical tab
`\a`	alert (bell)	`\0`	null character	`\\`	backslash
`\'`	single quote	`\"`	double quote	`\?`	question mark

A backslash followed by an ‘`x`’, ‘`d`’, or ‘`o`’ specifies the character with the corresponding numeric encoding in the ASCII character set:

`\x`*nn*	hexadecimal escape
`\d`*nnn*	decimal escape
`\o`*nnn*	octal escape

There must be exactly two hexadecimal digits for hexadecimal character escapes, and exactly three digits for decimal and octal character escapes. Illegal hexadecimal, decimal, and octal numbers, as well as character escapes specifying ASCII values greater than 255 will cause a lexical error.

For example, the following characters all denote the carriage return character (ASCII code 13):

```
\r        \x0d       \d013      \o015
```

A long string may be broken into multiple lines by preceding each newline with a backslash. Such escaped newlines are ignored during formation of the string constant.

[†] When typing strings in, the graphical user interface accepts multi-line strings, but the character-based read-eval-print loop does not.

A backslash followed by any other character than those listed above will cause a lexical error.

Productions:

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*

2.4.8 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"`

Appendix 2.A 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.

Appendix 2.B 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 ‘"’

Appendix 2.C 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*

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

keyword-slot	→	*small-keyword* {*cap-keyword*} ‘=’ *regular-object* | *small-keyword identifier* {*cap-keyword identifier*} ‘=’ *regular-object*
slot-name	→	*identifier* | *parent-name*
parent-name	→	*identifier* ‘*’

Appendix 2.D 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 [2.1.7].

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*.

3 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].

3.1 World Organization

3.1.1 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`).

3.1.2 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.)

3.2 The Roots of Behavior

3.2.1 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

3.2.2 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

3.2.3 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`.

3.2.4 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

3.3 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 the SELF 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 stuctures in SELF except message sends, returns, and VM error handling are implemented using blocks.

3.3.1 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

3.3.2 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

3.3.3 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

3.3.4 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

3.4 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

3.4.1 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

3.4.2 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

3.5 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)

3.5.1 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

3.5.2 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 an 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

3.5.3 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 addtions 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

3.5.4 Tree-Based Sets and Dictionaries

```
collection
    tree
        treeNodes abstract
            treeNodes bag
            treeNodes 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

3.5.5 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

3.5.6 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

3.6 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

3.7 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—itself.) 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

3.8 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 used `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

3.9 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 manyUnix shells.

Modules: process, scheduler, semaphore, prompt, history

3.10 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 the VM Reference Manual. 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 `foreign-Code` 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:

`linker`	dynamic linker for current platform
`sunLinker`	dynamic linker implementation for SunOS/Solaris
`foreignCodeDB`	registry for `foreignCode` objects

Modules: foreign

3.11 I/O and Unix

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

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:

`stdin, stdout, stderr`	standard Unix streams
`tty`	console terminal capabilities

Modules: unix, stdin, tty, ttySupport, termcap

3.12 Other Objects

Here are some interesting oddball objects not discussed elsewhere:

`comparator`	an object that can compute “diffs” between sequences
`compilerProfiling`	compiler profiling
`desktop`	The controlling object for the graphical user interface
`history`	A history of commands typed at the prompt, and their results
`memory`	memory system interface (GC, snapshot, low space, etc.)
`monitor`	system monitor (spy) control
`nil`	indicates an uninitialized value
`platforms`	possible hardware platforms
`preferences`	user configuration preferences
`profiling`, `flatProfiling`	controls SELF code profiling
`prompt`	interactive read-eval-print loop
`scheduler`	SELF process scheduler
`snapshotAction`	actions to do before/after a snapshot
`thisHost`	describes the current host platform
`times`	reports user, system, cpu, or real time
`typeSizes`	bit/byte sizes for primitive types
`vmProfiling`	virtual machine profiling

3.13 How to build the world

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 provided with your distribution (in `Optional.SelfSource.tar.Z`). You can also do this after writing the world out using the transporter (`transporter fileOut fileOutAll`).

To create the default object world:

1. Start the SELF VM:

   ```
   % Self
   Self Virtual Machine Version 4.0.2, Thu 09 Feb 95 19:41:30
   Copyright 1989-95: 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 that because the world is empty, you must use the primitive to do this.

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

   ```
   VM# 'all2.self' _RunScript
   ```

   Various configurations are possible: `all2` is the released system; `smallUI2` is the same but without the various example applications; and `all` contains the old (release 3.0) experimental user interface.

   Unless you have asked SELF not to print script names, you should see something like:

   ```
   reading all2.self
   reading init.self
   . . .
   ```

4. 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.

3.14 How to use the low-level interrupt facilities

There are two low-level ways to interrupt a running SELF program[†], 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:

```
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

[†] Normally, you would use debugging facilities provided in the programming environment.

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.

3.15 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
attach: n	attach the process with object reference number n
detach	detach the debugged process
step[:n]	execute (n) non trivial bytecodes[a]
stepi[:n]	execute (n) bytecodes
next[:n]	execute (n) non trivial bytecodes in the current activation
nexti[:n]	execute (n) bytecodes in the current activation
finish	finish executing the current activation
cont	continue execution
trace	print out a stack trace of the process
show	display the current activation
show: n	go to and display the nth activation on the stack
status	display the status of the debugged process
up[: n]	go up (n) activation(s)
upLex	go up to the lexical enclosing scope of this activation
down[: n]	go down (n) activation(s)
lookup: <name>	lookup the given name in the context of the current activation

a. 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.

Appendix 3.A Glossary of Useful Selectors

This glossary lists some useful selectors. It is by no means exhaustive.

Copying

clone	shallow copy (for use within an object; clients should use copy)
copy	copy the receiver, possibly with embedded copies or initialization

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

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
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

Bitwise operations (integers)

&&	and
||	or
^^	xor
complement	bitwise complement
<<	logical left shift
>>	logical right shift
<+	arithmetic left shift
+>	arithmetic right shift

Logical operations (booleans)

&&	and
||	or
^^	xor
not	logical complement

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

Printing

print	print object on `stdout`
printLine	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

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

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
removeAllOccurences:	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 occurences 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

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

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

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*

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)

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

Virtual Machine-Generated

Errors

undefinedSelector:Type:Delegatee:MethodHolder:Arguments:	lookup found no matching slot
ambiguousSelector:Type:Delegatee:MethodHolder:Arguments:	lookup found more than one matching slot
missingParentSelector:Type:Delegatee:MethodHolder:Arguments:	parent slot through which resend was delegated was not found
performTypeErrorSelector:Type:Delegatee:MethodHolder:Arguments:	first argument to the `_Perform` primitive was not a canonical string
mismatchedArgumentCountSelector:Type:Delegatee:MethodHolder:Arguments:	number of args supplied to `_Perform` 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

4 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.

4.1 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 that 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.

4.2 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.

4.2.1 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.

4.2.2 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.

4.2.3 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 4.4. 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.

4.3 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 4.3.2 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*.

4.3.1 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.

4.3.2 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.

4.4 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.
       ...
```

4.5 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]
```

4.6 `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:`).

4.7 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.

4.8 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 indistinguisability 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.

5 Virtual Machine Reference

5.1 Startup options

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

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

These options are provided for use by SELF VM implementors:

-F	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.
-l logfile	Writes a log of events generated by the spy to *logfile*.
-r	Disables real timer interrupts
-t	Disables all timers

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

5.2 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 section 5.3.

5.3 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.

5.4 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.

5.5 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.

5.6 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 Manual.

- `No ’foo’ slot found in shell <0>.`
  The lookup found no slot matching the selector `foo`.
- `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. Ambiguities can often be resolved by changing parent priorities.
- `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.

5.7 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.

5.8 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 in 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.

5.9 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 call a “non-LIFO” block. Non-LIFO blocks are not supported by the current version of SELF.

5.10 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).

5.11 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 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*)

.

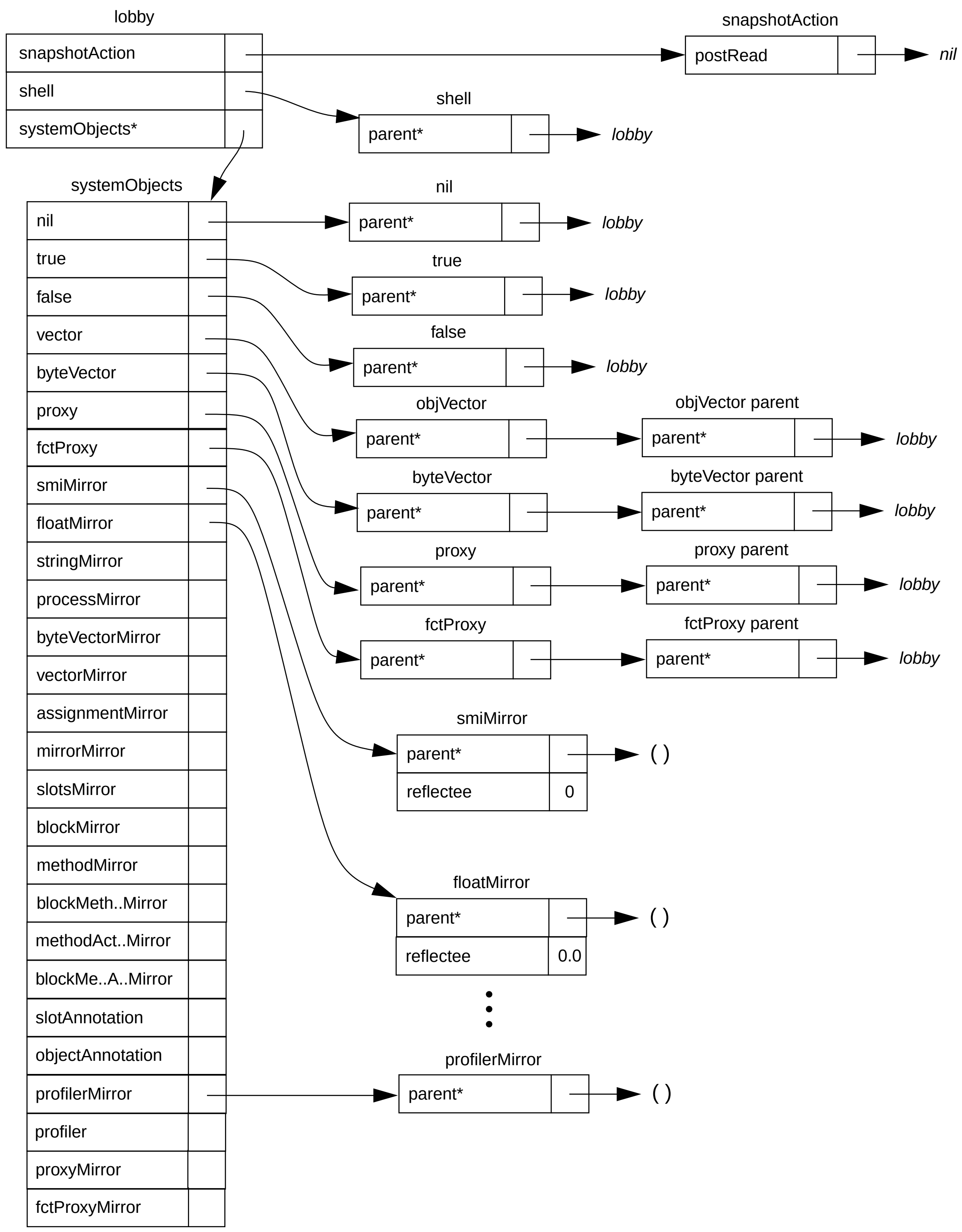


Figure 3 The initial SELF world (part 1)

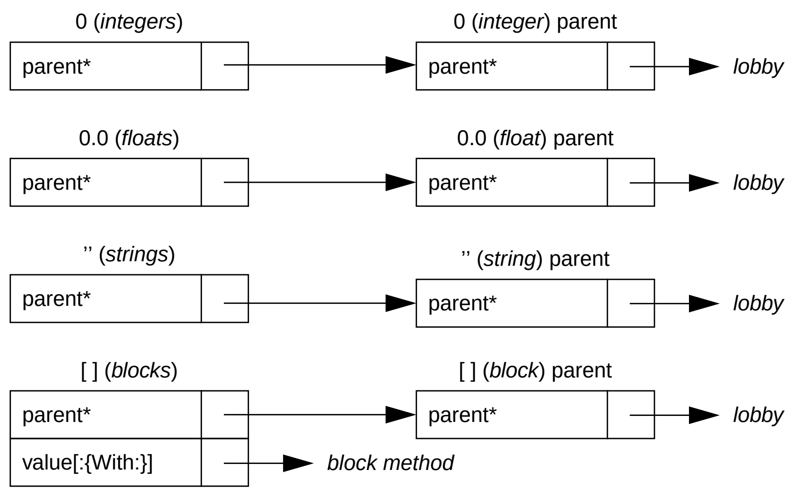


Figure 4 The initial SELF world (part 2)

Table 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 `_RunScript`, or used as the initializers of slots, are evaluated.
Objects in the lobby	
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 section 5.2), `postRead`. This slot initially contains `nil`.
systemObjects	This object contains slots containing the general system objects, including `nil`, `true`, `false`, and the prototypical vectors and mirrors.
Objects in `systemObjects`	
nil	The initializer for slots that are not explicitly initialized. Indicates “not a useful object.”
true	Boolean true. Argument to and returned by some primitives.
false	Boolean false. Argument to and returned by some primitives.
vector	The prototype for (normal) vectors.
byteVector	The prototype for byte vectors.
proxy	The prototype for proxy objects.
fctProxy	The prototype for `fctProxy` objects.
vector parent	The object that `vector` inherits from. Since all object vectors will inherit from this object (because they are cloned from `vector`), this object will be the repository for shared behavior (a *traits object*) for vectors.
byteVector parent	Similar to `vector parent`: the byteVector traits object.

slotAnnotation	The default slot annotation object.
objectAnnotation	The default object annotation object
profiler	The prototype for profilers.
mirrors	See below.

Literals and their parents

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

Prototypical mirrors

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 Figure 5 by "`( )`").

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

5.12 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 2; other option primitives can be found in Appendix 5.B, and a list of all option primitives and their current settings can be printed with the primitive `_PrintOptionPrimitives`.

Table 2 Some useful option primitives

Name	Description
_PrintPeriod[:][†]	Print a period when reading a script file with `_RunScript`. Default: `false`.
_PrintScriptName[:]	Print the file name when reading a script file. Default: `false`.
_Spy[:]	Start the system monitor (see Appendix 5.A for details). Default: `false`.
_StackPrintLimit[:]	Controls the number of stack frames printed by `_PrintProcessStack`. Default: 20.
_DirPath[:]	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 argument 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.

[†] 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.

5.13 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.

5.13.1 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.

5.13.2 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. Section 5.13.5 describes C glue and section 5.13.9 describes C++ glue.

5.13.3 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 section 5.13.16 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.

[†] Note that many libraries are already included in the SELF virtual machine (e.g. libc.a) and hence should not be added to the dynamic library.

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 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, 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.

5.13.4 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;
}
```

[†] If you try this example, be sure to type in all the “double” commas - they are necessary because of technical details with C++ macros.

```
/* 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.

5.13.5 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 5.13.9). The following sections describe each of these parts of C glue.

5.13.6 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 5.13.4:

```
    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 3 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 3 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.

A word of warning: after calling `failure` or `unix_failure` a normal `return` must be done. The value returned (in the example `NULL`) is ignored.

† The `any` conversion is the lone exception: it takes *two* SELF objects and produces *one* C argument.

5.13.7 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 3.
- `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 3, specifies how to do this conversion.

5.13.8 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 3 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 3 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 3, specifies how to do this conversion.

5.13.9 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.

5.13.10 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 5.13.6. 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 3), 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 3 for a description of all argument conversions. In the sockaddr example, there are no arguments.

5.13.11 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 3 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.

† `sockaddr_in` is actually not a C++ class, but a C struct. However, C++ treats structs and classes the same.

5.13.12 Calling member functions

Calling member functions is similar to calling “plain” functions, so please also refer to section 5.13.6. 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 3, 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 5.13.6 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 3 may be used.

5.13.13 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).

5.13.14 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:

[†] In fact there is no such member function defined on `sockaddr_in` objects.

- 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 3 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.

Table 3 : 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	1
signed_char		smallInt	signed char	badTypeError overflowError	
unsigned_char		smallInt	unsigned char	badSignError badTypeError overflowError	
short		smallInt	short	badTypeError overflowError	
signed_short		smallInt	signed short	badTypeError overflowError	
unsigned_short		smallInt	unsigned short	badSignError badTypeError overflowError	
int		smallInt	int	badTypeError	
signed_int		smallInt	signed int	badTypeError	
unsigned_int		smallInt	unsigned int	badSignError badTypeError	
long		smallInt	long	badTypeError	
signed_long		smallInt	signed long	badTypeError	

[†] The `any` conversion is the only conversion that has more than one incoming object.

Table 3 : Argument conversions - from SELF to C/C++

Conversion	Second part	SELF type	C/C++ type	Errors	Notes
unsigned_long		smallInt	unsigned long	badSignError	
smi		smallInt	smi	badTypeError	2
unsigned_smi		smallInt	smi	badSignError badTypeError	2
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 badTypeError	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 badTypeError	4, 5, 6
cbv	ptr_type	byte vector	ptr_type	badTypeError	7
cbv_len	ptr_type	byte vector	ptr_type, int	badSizeError badTypeError	7
cbv_null	ptr_type	byte vector/0	ptr_type	badTypeError	7
cbv_len_null	ptr_type	byte vector/0	ptr_type, int	badSizeError badTypeError	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 badTypeSealError deadProxyError, nullPointerError	9
proxy_null	(ptr_type, type_seal)	proxy	ptr_type	badTypeError badTypeSealError deadProxyError	9
any_oop		any object	oop		10

Table 3 : Argument conversions - from SELF to C/C++

Conversion	Second part	SELF type	C/C++ type	Errors	Notes
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 deadProxyError	12

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 pro-

duce 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.

5.13.15 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 4 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 4 : 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	

Table 4 : Result conversions - from C/C++ to SELF

Conversion	Second part	C/C++ type	SELF type	Errors	Notes
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	nullPointerError	3
proxy	(ptr_type, type_seal)	ptr_type	proxy	nullPointerError	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	nullPointerError	3, 6, 8
fct_proxy_null	(ptr_type, type_seal, arg_count)	ptr_type	fctProxy		6, 8
oop		oop	corresponding object		7, 8

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 `nullPointerError` 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.

5.13.16 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 Figure 5 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 Figure 5 shows a two stage pipe line. Here stdout of the first process is connected to stdin of the second process.

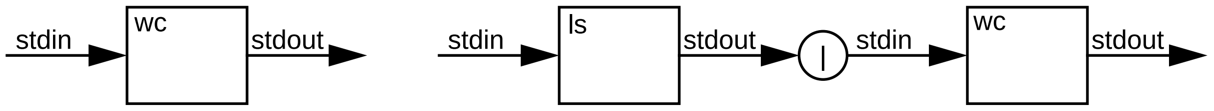


Figure 5. A single UNIX process and an pipe line

Figure 5 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 Figure 5 shows how the UNIX/SELF process pair can fit seamlessly into a pipe line.

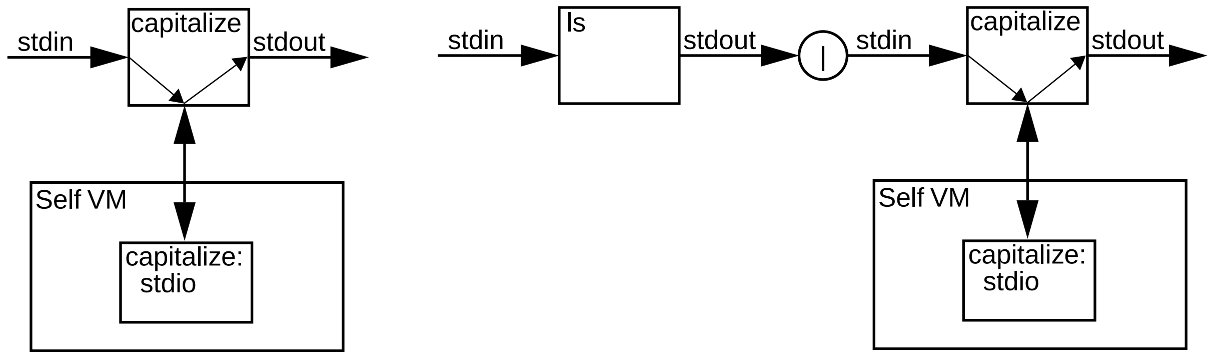


Figure 6. A SELF process and how it fits into a pipe line

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.

5.13.17 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).

5.13.18 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 forked process executes the method `server handleRequest` which:

   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.

5.13.19 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`.

5.13.20 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.
  ).
```

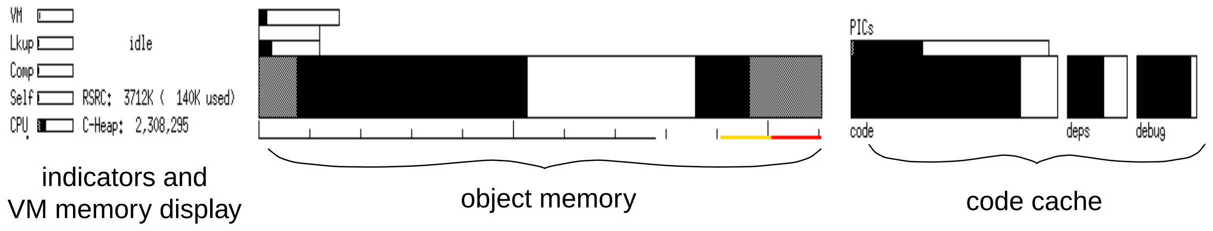
This method copies a `socket` object and returns the copy. The local slot `sbl` is initialized to a `foreignFct` object. The body of the method simply sends `value:With:` to the `foreignFct` object. The first argument is the port number to request for the socket, the second argument is a `deadCopy` of 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 5.13.1).

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`.

Appendix 5.A 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 is 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 5 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-inling 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 6 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 7 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 colelction 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.

[†] The segregation of (the vector of bytes in) byte vectors from other objects is an implementation detail improving scavenging and scanning performance (see [Lee88] and [CUL89] for details).

Color	**Meaning**
black	Allocated, residing in real memory.
gray	Allocated, paged out.[†]
white	Unallocated memory.

[†] The residency information is updated only once a second for efficiency reasons; all other information is updated continuously. Also, the gray area does not indicate *what* is paged out, only *how much*.

Appendix 5.B 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.

5.B.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 8 Primitive failures

Prefix	Description
primitiveNotDefinedError	Primitive not defined.
primitiveFailedError	General primitive failure (for example, an argument has an invalid value).
badTypeError	The receiver or an argument has the wrong type.
badTypeSealError	Proxy's type seal did not match expected type seal.
divisionByZeroError	Division by zero.
overflowError	Integer overflow. This can occur in integer arithmetic primitives or in UNIX (when the result is too large to be represented as an integer).
badSignError	Integer receiver or argument has wrong sign.
alignmentError	Bad word alignment in memory.
badIndexError	The vector index (e.g. in `_At:`) is out of bounds (too large or negative).

badSizeError	An invalid size of a vector was specified, e.g. attempting to clone a vector with a negative size (see `_Clone:Filler:` and `_CloneBytes:Filler:` below).
reflectTypeError	A mirror primitive was applied to the wrong kind of slot, e.g. `_MirrorParentGroupAt:` to a slot that isn’t a parent slot.
outOfMemoryError	A primitive could not complete because its results would not fit in the existing space
stackOverflowError	The stack overflowed during execution of the primitive or program.
slotNameError	Illegal slot name.
argumentCountError	Wrong number of arguments.
unassignableSlotError	This slot is not assignable.
lonelyAssignmentSlotError	Assignment slot must have a corresponding data slot.
parallelTWAINSError	Can not invoke TWAINS primitive (another process is already using it).
noProcessError	This process does not exist.
noActivationError	This method activation does not exist.
noReceiverError	This activation has no receiver.
noParentSlot	This activation has no lexical parent.
noSenderSlot	This activation has no sender slot.
deadProxyError	This proxy is dead and can not be used.
liveProxyError	This proxy is live and can not be used to hold a proxy result.
wrongNoOfArgsError	Wrong number of arguments was supplied with call of foreign function.
nullPointerError	Foreign function returned null pointer.
nullCharError	Can not pass byte vector containing null char to foreign function expecting a string.
prematureEndOfInputError	Premature end of input during parsing.
noDynamicLinkerError	Primitive depends on dynamic linker which is not available in this system.
EPERM, ENOENT, ...	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 `/usr/include/sys/errno.h`; 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.

5.B.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)

Appendix 5.C Getting the optional source files

In addition to the main part of the release, there are three other files available for ftp. However, most users will not need these files, and since they will complicate the installation, we ***strongly discourage*** getting these files unless you are *sure* you will need them. You will be able to run the SELF system without any of these optional files.

If you do need any of these files, ftp to `self.smli.com` and use login name `optional` and password `optional` (do not use anonymous ftp). Go to the directory `/optional/4.0`..

Archive file	Contents
`Optional.Self-Source.tar.Z`	Contains the default SELF world of objects, as emitted by the transporter (i.e., SELF source for all the system objects). Since these files can be reconstituted by the transporter, *you should not bother with this file* unless you want to rebuild a SELF world from source code (these files are not intended for human consumption).
`Optional.Glue.tar.Z`	Contains the header files necessary to write the “glue code” that allows SELF programs to call C/C++ functions. *You should not bother with this file* unless you plan to extend SELF by gluing in external libraries.
`Optional.VM.tar.Z`	Contains source code for the SELF virtual machine, roughly 80,000 lines of C++. An installation of GNU g++ version 2.6.0 is required to compile this code. *You should not bother with this file* unless you want to study or change the virtual machine. VM documentation is almost non-existent.

6 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.

[Age94b] Ole Agesen. Concrete Type Inference: Delivering Object-Oriented Applications. Technical Report, Sun Microsystems Labs, SMLI TR-96-52, 1996.

[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.

Index