

Little Smalltalk Users Manual - Version Three

Tim Budd

Department of Computer Science
Oregon State University
Corvallis, Oregon
97331 USA

ABSTRACT

Version three of Little Smalltalk was designed specifically to be easy to port to new machines and operating systems. This document provides the basic information needed to use Version Three of Little Smalltalk, plus information needed by those wishing to undertake the job of porting the system to a new operating environment.

The first version of Little Smalltalk, although simple, small and fast, was in a number of very critical ways very Unix specific. Soon after the publication of the book *A Little Smalltalk*, requests started flooding in asking if there existed a port to an amazingly large number of different machines, such as the IBM PC, the Macintosh, the Acorn, the Atari, and even such systems as DEC VMS. Clearly it was beyond our capabilities to satisfy all these requests, however in an attempt to meet them partway in the summer of 1988 I designed a second version of Little Smalltalk, which was specifically designed to be less Unix specific and more amenable to implementation of different systems.

This document describes is divided into two parts. In part one I describe the

basic features of the user interface. This is essential information for anybody wishing to use the system. In part two we give the basic information needed by anybody wishing to undertake the task of porting version three Little Smalltalk to a new machine.

1. Getting Started

How you get started depends upon what kind of system you are working on. Currently there are two styles of interface supported. A line-oriented, tty style stdin interface is available, which runs under Unix and other systems. There is also a window based system which runs under X-windows and on the Mac.

1.1. The stdin/stdout interface

Using the stdin/stdout interface, there is a prompt (the ``>" character) typed to indicate the system is waiting for input. Expressions are read at the keyboard and evaluated following each carriage return. The result of the expression is then printed.

```
> 5 + 7
12
```

Global variables can be created simply by assigning to a name. The value of an assignment statement is the value of the right hand side.

```
x <- 3
3
```

Multiple expressions can appear on the same line separated by periods. Only the last expression is printed.

```
y <- 17. 3 + 4
7
```

1.2. The windowing interface

The windowing interface is built on top of guido van rossums standard window package, and runs on top of systems that support standard windows. These include X-11 and the Macintosh.

When you start up the system, there will be a single window titled ``workspace". You can enter expressions in the workspace, then select either the menu items ``do it" or ``print it". Both will evaluate the expression; the latter, in addition, will print the result.

A number of other menu commands are also available. These permit you to save the current image, exit the system, or start the browser.

The browser is an interface permitting you to easily view system code. Selecting a class in the first pane of the browser brings up a second pane in which you can select methods, selecting a method brings up a third pane in which you can view and edit text. Selecting ``compile" following the editing of text will attempt to compile the method. If no errors are reported, the method is then available for execution.

2. Exploring and Creating

This section describes how to discover information about existing objects and create new objects using the Little Smalltalk system (version three). In Smalltalk one communicates with objects by pass-ing messages to them. Even the addition message + is treated as a message passed to the first object 5, with an argument represented by the second object. Other messages can be used to discover information about various objects. The most basic fact you can discover about an object is its class. This is given by the message class, as in the following examples:

```
> 7 class  
Integer  
> nil class  
UndefinedObject
```

Occasionally, especially when programming, one would like to ask whether the class of an object matches some known class. One way to do this would be to use the message `==`, which tells whether two expressions represent the same object:

```
> ( 7 class == Integer)
True
> nil class == Object
False
```

An easier way is to use the message `isMemberOf:`:

```
> 7 isMemberOf: Integer
True
> nil isMemberOf: Integer
False
```

Sometimes you want to know if an object is an instance of a particular class or one of its sub-classes; in this case the appropriate message is `isKindOf:`.

```
> 7 isMemberOf: Number
False
> 7 isKindOf: Number
True
```

All objects will respond to the message `display` by telling a little about themselves. Many just give their class and their printable representation:

```
> 7 display
(Class Integer) 7
> nil display
(Class UndefinedObject) nil
```

Others, such as classes, are a little more verbose:

```
> Integer display
Class Name: Integer
SuperClass: Number
Instance Variables:
no instance variables
Subclasses:
```

The display shows that the class `Integer` is a subclass of class `Number` (that is, class

Number is the superclass of Integer). There are no instance variables for this class, and it currently has no subclasses. All of this information could be obtained by means of other messages, although the display form is the easiest. [Note: at the moment printing subclasses takes a second or two. I'm not sure why.]

```
> List variables display
links
> Integer superClass
Number
> Collection subclasses display
IndexedCollection
Interval
List
```

About the only bit of information that is not provided when one passes the message display to a class is a list of methods the class responds to. There are two reasons for this omission; the first is that this list can often be quite long, and we don't want to scroll the other information off the screen before the user has seen it. The second reason is that there are really two different questions the user could be asking. The first is what methods are actually implemented in a given class. A list containing the set of methods implemented in a class can be found by passing the message methods to a class. As we saw with the message subclasses shown above, the command display prints this information out one method to a line:

```
> True methods display
#ifTrue:ifFalse:
#not
```

A second question that one could ask is what message selectors an instance of a given class will respond to, whether they are inherited from superclasses or are defined in the given class. This set is given in response to the message respondsTo. [NOTE: again for some reason I'm not sure of this command seems to take a long time to execute].

```
> True respondsTo display
#class
#==
#hash
#isNil
#display
#=
#basicSize
#isMemberOf:
#notNil
#print
#basicAt:put:
#isKindOf:
#basicAt:
#printString
#or:
#and:
#ifFalse:ifTrue:
#ifTrue:
#ifFalse:
#not
#ifTrue:ifFalse:
```

Alternatively, one can ask whether instances of a given class will respond to a

specific message by writing the message selector as a symbol:

```
> String respondsTo: #print
True
> String respondsTo: #+
False
```

The inverse of this would be to ask what classes contain methods for a given message selector. Class Symbol defines a method to yield just this information:

```
> #+ respondsTo display
Integer
Number
Float
```


The method that will be executed in response to a given message selector can be displayed by means of the message viewMethod:

```
> Integer viewMethod: #gcd:
gcd: value
    (value = 0) ifTrue: [ ^ self ].
    (self negative) ifTrue: [ ^ self negated gcd: value ].
    (value negative) ifTrue: [ ^ self gcd: value negated ].
    (value > self) ifTrue: [ ^ value gcd: self ].
    ^ value gcd: (self rem: value)
```

Some Smalltalk systems make it very difficult for you to discover the bytecodes that a method gets translated into. Since the primary goal of Little Smalltalk is to help the student to discover how a modern very high level language is implemented, it makes sense that the system should help you as much as possible discover everything about its internal structure. Thus a method, when presented with the message display, will print out its bytecode representation.

```
> Char methodName: #isAlphabetic ; display
Method #isAlphabetic
    isAlphabetic
        ^ (self isLowercase) or: [ self isUppercase ]
```

```
literals
Array ( #isLowercase #isUppercase )
bytecodes
32 2 0
129 8 1
144 9 0
250 15 10
9 0 9
32 2 0
129 8 1
145 9 1
242 15 2
245 15 5
241 15 1
```

Bytecodes are represented by four bit opcodes and four bit operands, with occasional bytes representing data (more detail can be found in the book). The three numbers written on each line for the bytecodes represent the byte value followed by the upper four bits and the lower four bits.

If you have written a new class and want to print the class methods on a file you can use the message `fileOut:`, after first creating a file to write to. Both classes and individual methods can be filed out, and several classes and/or methods can be placed in one file. [NOTE - file out doesn't work yet].

```
> f <- File new
> f name: 'foo.st'
> f open: 'w'
> Foo fileOut: f
> Bar fileOut: f
> Object fileOutMethod: #isFoo to: f
> f close
```

The file ``newfile" will now have a printable representation of the methods for the class Foo. These can subsequently be filed back into a different smalltalk image.

```
> f <- File new
> f name: 'foo.st'
> f open: 'r'
> f fileIn
> 2 isFoo
False
```

Finally, once the user has added classes and variables and made whatever other changes they want, the message saveImage, passed to the pseudo variable smalltalk, can be used to save an entire object image on a file. If the writing of the image is successful, a message will be displayed.

```
> smalltalk saveImage
Image name? newimage
image newimage created
>
```

Typing control-D causes the interpreter to exit.

When the smalltalk system is restarted, an alternative image, such as the image just created, can be specified by giving its name on the argument line:

```
st newimage
```

Further information on Little Smalltalk can be found in the book.

3. New Methods, New Classes

3.1. Stdin/Stdout Interface

New functionality can be added using the message addMethod. When passed to an instance of Class, this message drops the user into a standard Unix Editor. A body for a new method can then be entered. When the user exits the editor, the method body is compiled. If it is syntactically correct, it is added to the methods for the class. If it is incorrect, the user is given the option of re-editing the method. The user is first prompted for the name of the group to which the method belongs.

```
> Integer addMethod
```

```
... drop into editor and enter the following text
% x
  ^ ( x + )
... exit editor
compiler error: invalid expression start )
edit again (yn) ?
...
```

In a similar manner, existing methods can be editing by passing their selectors, as symbols to the message editMethod:.

```
> Integer editMethod: #gcd:
... drop into editor working on the body of gcd:
```

The name of the editor used by these methods is taken from a string pointed to by the global variable editor. Different editors can be selected merely by redefining this value:

```
editor <- 'emacs'
```

Adding a new subclass is accomplished by sending the message `addSubClass:instanceVariableNames:` to the superclass object. The first argument is a symbol representing the name, the second is a string containing the names of any instance variables.

```
> Object addSubClass: #Foo instanceVariableNames: 'x y'
Object
  Foo display
Class Name: Foo
SuperClass: Object
Instance Variables:
x
y
```

Once defined, `addMethod` and `editMethod:` can be used to provide functionality for the new class.

New classes can also be added using the `fileIn` mechanism.

3.2. The Windowing Interface

Using the windowing interface, new classes are created by selecting the menu item `add class` in the first browser window. New Methods are selected by choosing `new method` in a subsequent window.

4. Incompatibilities with the Book

It is unfortunately the case that during the transition from version 1 (the version described in the book) and version 3, certain changes to the user interface were required. I will describe these here.

The first incompatibility comes at the very beginning. In version 1 there were a great number of command line options. These have all been eliminated in version three. In version three the only command line option is the file name of an image file.

The interface to the editor has been changed. In version one this was handled by the system, and not by Smalltalk code. This required a command format that was clearly not a Smalltalk command, so that they could be distinguished. The convention adopted

was to use an APL style system command:

```
)e filename
```

In version three we have moved these functions into Smalltalk code. Now the problem is just the reverse, we need a command that is a Smalltalk command. In addition, in version one entire classes were edited at once, whereas in version three only individual methods are edited. As we have already noted, the new commands to add or edit methods are as follows:

```
classname addMethod  
classname editMethod: methodname
```

The only other significant syntactic change is the way primitive methods are invoked. In version one these were either named or numbered, something like the following:

```
<primitive 37 a b>  
<IntegerAdd a b>
```

In version three we have simply eliminated the keyword primitive, so primitives now look like:

```
<37 a b>
```

There are far fewer primitives in version three, and much more of the system is now performed using Smalltalk code.

In addition to these syntactic changes, there are various small changes in the class structure. I hope to have a document describing these changes at some point, but as of right now the code itself is the best description.

5. Implementors Information

The remainder of this document contains information necessary for those wishing to examine or change the source code for the Little Smalltalk system.

5.1. Finding Your Way Around

In this section we describe the files that constitute version three of the Little Smalltalk system.

memory.c

This is the memory manager, the heart of the Little Smalltalk system. Although it

uses a straight-forward reference counting scheme, a fair amount of design effort has gone into making it as fast as possible. By modifying it's associated description file (memory.h) a number of operations can be specified either as macros or as function calls. The function calls generally perform more error checking, and should be used during initial development. Using macros, on the other hand, can improve performance dramatically. At some future date we hope to make available both reference counting and garbage collection versions of the memory manager.

names.c

The only data structures used internally in the Little Smalltalk system are arrays and name tables. A name table is simply an instance of class Dictionary in which keys are symbols. Name tables are used to implement the dictionary of globally accessible values, symbols, and to implement method tables. This module provides support for reading from name tables.

news.c

This module contains several small utility routines which create new instances of various standard classes.

interp.c

This module implements the actual bytecode interpreter. It is the heart of the system, where most execution time is spent.

primitive.c

This module contains the code that is executed to perform primitive operations. Only the standard primitives (see the section on primitives) are implemented in this module. File primitives and system specific primitives are implemented in another module, such as `unixio.c` for the Unix system and `macio.c` for the Macintosh version.

unixio.c, filein.c

These two modules contain I/O routines.

lex.c, parser.c

The files `lex.c` and `parser.c` are the lexical analyzer and parser, respectively, for compiling the textual representation of methods into bytecodes. In the current version parsing is done using a simple (although large) recursive descent parser.

st.c

The file `st.c` is the front end for the Unix version of Little Smalltalk. On the Macintosh version it is replaced by the pair of files `macmain.c` and `macevent.c`.

initial.c

This module contains code that reads the module form of Smalltalk code, creating an object image. This is not part of the Smalltalk bytecode interpreter, but is used in building the initial object image (see next section).

There are description files (`.h` files, in standard C convention) which describe many of the modules described above. In addition, there is a very important file called `env.h` (for ``environment"). This file describes the characteristics of the operating system/machine you are running on. The general structure of this file is that the user provides one definition for their system, for example

```
# define LIGHTC
```

to indicate using the Lightspeed C compiler on the macintosh, for example. Following this are block of code which, based on this one definition, define other terms representing the specific attributes of this system. Where ever possible new code should be surrounded by `ifdef` directives based on words defined in this manner. The next section describes this in more detail.

5.2. Defining System Characteristics

There are many ways in which compilers and operating systems differ from each other. A fair amount of work has been expended in making sure the software will operate on most machines, which requires that different code fragments be used on different systems. In large part these are controlled by a single ``meta-define" in the file `env.h`. Setting this one value then causes the expansion of another code segment, which then defines many more options.

In the event that you are attempting to port the software to a system that has not previously been defined, you will need to decide which set of options to enable. The next two sections contain information you may need in making this determination.

Define Options

Many options are specified merely by giving or not giving a `DEFINE` statement in the file `env.h`. The following table presents the meaning for each of these values:

ALLOC

Defined If there is an include file called `alloc.h` which defines `calloc`, `malloc`, and the like.

BINREADWRITE

Defined if the fopen specification for binary files must include the "b" modifier. This is true on many MS-DOS inspired systems.

NOENUMS

Defined if enumerated datatypes are not supported. If defined, these will be replaced by #define constants.

NOTYPEDEF

Defined if the typedef construct is not supported. If defined, these will be replaced by #define constructs.

NOVOID

Defined if the void keyword is not recognized. If defined, expect lint to complain a lot about functions returning values which are sometimes (or always) ignored.

SIGNALS

Used if both the <signals.h> package and the <longjmp.h> package are available, and if the routine used to set signals is signal. Incompatible with SSIGNALS.

SSIGNALS

Used if both the <signals.h> package and the <longjmp.h> package are available, and if the routine used to set signals is ssignal. Incompatible with SIGNALS.

STRING

Used if the string functions (strcpy, strcat and the like) are found in <string.h>. This switch is incompatible with STRINGS.

STRINGS

Used if the string functions (strcpy, strcat and the like) are found in <strings.h>. This switch is incompatible with STRING.

In addition, several routines can optionally be replaced by macros for greater efficiency. See the file memory.h for more information.

5.3. Building an Initial Object Image

There are two programs used in the Little Smalltalk system. The first is the actual bytecode interpreter. The use of this program is described in detail in other documents (see ``Exploring and Creating"). The Little Smalltalk system requires, to start, a snapshot representation of memory. This snapshot is called an object image, and the purpose of the second program, the initial object image maker, is to construct an initial object image. In theory, the this program need only be run once, by the system administrator, and thereafter all users can access the same standard object image.

The object image format is binary. However, since the format for binary files will undoubtedly differ from system to system, the methods which will go into the initial image are distributed in textual form, called module form. Several modules are combined to create an object image. The following describes the modules distributed on the standard tape, in the order they should be processed, and their purposes.

basic.st

This module contains the basic classes and methods which should be common to all implementations of Little Smalltalk.

mag.st

This module contains methods for those objects having magnitude, which are the basic subclasses of Magnitude.

collect.st

This module contains methods for the collection subclasses.

file.st

This module contains the classes and methods used for file operations. Although all implementations should try to support these operations, it may not always be possible on all systems.

unix.st

This module contains unix - specific commands, which may differ from those used under other operating systems.

mult.st

This module contains code for the multiprocessing scheduler.

init.st

This module contains code which is run to initialize the initial object image. These methods disappear after they have been executed. (or should; they don't really yet).

test.st

This file contains various test cases.

5.4. Object Memory

There are several datatypes, not directly supported by C, that are used in the Little Smalltalk system. The first of these is the datatype byte. A byte is an eight bit unsigned (hence positive) quantity. On many systems the appropriate datatype is unsigned char, however on other systems this declaration is not recognized and other forms may be required. To aid in converting to and from bytes the macro `byteToInt()` is used, which converts a byte value into an integer. In addition, the routines `byteAt` and

byteAtPut are used to get and put bytes from byte strings.

The other datatype is that used to represent object points. On most machines in which a short is 16 bits, the datatype short should suffice. Much more information on the memory module can be found in the file memory.h.

5.5. The Bottom End

The opposite extreme from the front end are those messages that originate within the Smalltalk bytecode interpreter and must be communicated to the user. We can divide these into two different classes of communications, editing operations and input/output operations. The following sections will treat each of these individually.

5.5.1. Editing

We have already mentioned that commands entered by the user are converted into methods, and passed to the same method compiler as all other methods. Before the user can create a new method, however, there must be some mechanism for allowing the user to enter the method.

One approach would be to read the method from the standard input, just as commands are read. While easy to implement, this approach would soon prove unsatisfactory, since for every error the user would need to reenter the entire method. So some form of update, or editing, must be provided. Again, the Unix interface and the Macintosh interface solve this problem in radically different ways.

5.5.1.1. Editing Under Unix

A request to edit or add a method is given by sending either the message `addMethod` or `editMethod:` to a class. The methods for these messages in turn call upon a common routine to perform the actual editing work.

```
addMethod
    self doEdit: "

editMethod: name
    self doEdit: ( methods at: name
                    ifAbsent: [ 'no such method ' print. ^ nil ] ) text

doEdit: startingText | text |
    text <- startingText.
    [ text <- text edit.
      (self addMethodText: text)
      ifTrue: [ false ]
      ifFalse: [ smalltalk inquire: 'edit again (yn) ? ' ]
    ] whileTrue
```

The Unix and MS-DOS versions of the system provide a method `edit` as part of the functionality of class `String`. When `edit` is passed to a string, an editing environment is established. The user performs editing tasks in that environment, and then exits the editing environment. Under Unix, this functionality is implemented using the file system.

```
edit | file text |
    file <- File new;
        scratchFile;
        open: 'w';
        print: self;
        close.
    (editor, ' ', file name) unixCommand.
    file open: 'r'.
    text <- file asString.
    file close; delete.
    ^ text
```

A file is created, and the contents of the string written to it. Then a standard Unix editor (given by the global variable `editor`) is invoked to process the file. After the user exits the editor, the contents of the file are read back as a string, the file is closed and deleted, and the string returned. The command `unixCommand` is implemented as a primitive, which invokes the `system()` system call:

unixCommand
^ <150 self>

Although the edit message is used by the system only for editing methods, it is general enough for any editing application and there is no reason why the user cannot use it for other purposes. By the way, the unixCommand message is also used to implement file deletes.

delete
('rm ', name) unixCommand

On MS-Dos systems this command should be changed to DEL.

5.5.1.2. Editing on the Macintosh

The Macintosh version takes an entirely different approach to the editing of methods. As in the Unix version, the user requests editing using the commands `editMethod:` and `addNewMethod`. And, as in the Unix version, these in turn invoke a common method.

```
addMethod
    self doEdit: ( self printString, ': new method') text: "

editMethod: name
    self doEdit: (self printString, ': ', name)
        text: (methods at: name
                ifAbsent: ['no such method' print. ^ nil ]) text
```

Here, however, when the user asks to edit a method, a new editing window is created.

```
doEdit: title text: text | w |
    w <- EditWindow new;
        acceptTask: [ self addMethodText: w getString ] ;
        title: title; create; print: text; showWindow
```

The edit window is initialized with the current text of the method. Thereafter, the user can edit this using the standard Macintosh cut and paste conventions. The user signifies they are satisfied with the result by entering the command `accept`, which causes the `acceptTask:` block to be executed. This block gets the text of the window (given by the message `getString`) and passes it to `addMethodText:`, which compiles the method, entering it in the method table if there are no errors.

5.5.2. Input/Output commands

Under the Unix system all input/output operations are performed using the file system and the global variables `stdin`, `stdout` and `stderr`. Thus the message `error:`, in class `Smalltalk`, merely prints a message to the standard error output and exits.

The macintosh version, although using the same file routines, does not have any notion of stan-dard input or standard output. Thus error messages (such as from `error:`) result in alert boxes being displayed.

There are also error messages that come from inside the `Smalltalk` interpreter itself. These are of two types, as follows:

1. System errors. These are all funnelled through the routine `sysError()`. System errors are caused by dramatically wrong conditions, and should generally cause the system to abort after printing the message passed as argument to `sysError()`.
2. Compiler errors. As we noted earlier, the method compiler is used to parse expressions typed directly at the keyboard, so these message can also arise in that manner. These are all funnelled through the routines `compilError()` and `compilWarn()`. These should print their arguments (two strings), in an appropriate location on the users screen. Execution continues normally after call.

5.6. Primitives

Primitives are the means whereby actions that cannot be described directed in Smalltalk are per-formed. In version three of the Little Smalltalk system, primitives are divided into three broad categories.

1. Primitives numbered less than 119 are all standard, and both the meaning and the implementation of these should be the same in all implementations of Little Smalltalk. These are largely just simple actions, such as mathematical operations.
2. Primitives numbered 120-139 are reserved for file operations. Although the meaning of these primitives should remain constant across all implementations, their implementation may differ.
3. Primitives number 150-255 are entirely implementation specific, and thus in porting to a new system the implementor is free to give these any meaning desired. For example under the Unix version there is, at present, only one such primitive, used to perform the system() call. On the other hand, the Macintosh version has dozens of primitives used to implement graphics functions, windowing function, editing and the like.

6. Distribution of New Implementations

The Little Smalltalk system is entirely public domain, and any user is free to redistribute it in any fashion they wish. As a service to the Smalltalk community, I would appreciate it if new implementors could send me a listing of changes they make, so that they can be incorporated into one standard distribution. Correspondence should be addressed to:

Tim Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331 USA

Copies of the most recent distribution can also be obtained by writing to this address. In mailing out distributions, there is a small charge for media and mailing costs.

7. New Features

If you type ``smalltalk echo" all input will be echoed (tty interface only). Typing

smalltalk echo again undoes this. This is useful for reading from scripts.