small

COLLABORATORS			
	TITLE : small		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 12, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

small

1.1 Amiga Little Smalltalk

1.2 Introduction

```
                          Introduction
```

(Timothy Budd)

This manual is intended as an aid in using the Amiga Little Smalltalk system. It is not intended to be used as an introduction to the Smalltalk language. Little Smalltalk is largely (with exceptions listed in a later section) a subset of the Smalltalk-80 language described in

Smalltalk blue

. A complete description of the classes included in the Amiga Little Smalltalk system and the messages they accept is given in

Class descriptions

.

Actually, the class descriptions here are from version 1 of Little Smalltalk. The right place to look for version 3 is in the running system!

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.

Amiga Notes
(David Faught)

I have been spending some of my hobby time enhancing Little Smalltalk, concentrating first on the programming environment. Since this has been mainly for my own enjoyment, I took the easy path and made these enhancements specific to my processor, a Commodore Amiga 2000 running AmigaDOS 2.1. A couple of years ago, I distributed an enhanced port of version 1 of Little Smalltalk with the beginnings of a GUI interface to the programming environment, and was planning out additions to the language itself for manipulation of GUI objects. Then I got a copy of version 3 of Little Smalltalk and dropped the version 1 project because version 3 already had a windowing GUI included. All I had to do was make it work on the Amiga! ;)

The first attempts at this used Guido Van Rossum’s Standard Windows software, of which I ported the "alfa" version to the Amiga. This was an attempt to keep even the GUI interface fairly platform independent. This worked, but just didn't give me what I wanted. Then, with some encouragement, I decided that platform independence isn't everything and used Stefan Stuntz's MagicUserInterface in place of Standard Windows. This worked much better and is actually much faster. There are several differences between the way this interface works and the way the original windowing interface from Timothy Budd works. Some day I may get around to documenting them, but in the meantime just double click and menu select away!

The documents that are distributed with Amiga Little Smalltalk have been reformatted to be used with AmigaGuide, which is a hypertext presentation tool now distributed with AmigaDOS. Although in most cases in the source code for Amiga Little Smalltalk, "#ifdef"s have been used to retain the original flexibility, this documentation has definitely turned the corner and is very much Amiga-specific. Many changes have been made to the original text with no notations delimiting Amiga extensions. If you would like the original documents, you will have to get them via anonymous ftp from ftp.cs.orst.edu, or the version 1 documents are on Fred Fish's disk number 37.

This AmigaGuide document is a conglomeration of Tim Budd's original version 1 and version 3 documents. It is mostly in sync with Little Smalltalk version 3, but there are no guarantees!

1.3 Acknowledgements

Acknowledgements

Little Smalltalk was developed by Timothy A. Budd and a group of his students at the University of Arizona in 1984. The original version of this manual followed in 1986. A more complete manual is
A Little Smalltalk
.

The Amiga port of version 1 of Little Smalltalk was done by Bill Kinnersley at Washington State University and widely distributed on Fred Fish's disk number 37.

The first Amiga port of version 3 of Little Smalltalk (that I know of) was done in 1993 by Dan Griffin.

MUI - MagicUserInterface Copyright © 1993-94 Stefan Stunz.

Standard Windows version 0.9.5 was written by Guido Van Rossum. Amiga port by David A. Faught, circa 1994.

The "Amiga-ized" version 1 of Little Smalltalk and this manual were done by David A. Faught in 1993. Version 3 in 1995.

* Smalltalk-80 is a trademark of the Xerox Corporation.

```
  * Unix is a trademark of Bell Laboratories.
```

1.4 Distribution

```
                       Distribution

     The Little Smalltalk system is public domain,  and  may
be  distributed  further  as  long  as proper attribution is
given in all published references.  The  Amiga  version   is
dependent on the MagicUserInterface,  which  is  not  public
domain.

     In the interests of keeping the distribution up to date
and  as  error  free  as  possible, we wish to keep track of
known sites using the system.  People  interested   should
contact  Timothy  Budd,   at  the  address  listed  below.
Changes,  modifications,  or improvements to the code or the
standard  library  can  be  submitted  also,  and  will be
considered for inclusion in future distributions.

  The Little  Smalltalk system  is distributed without
responsibility for the performance of the system and without
any guarantee of maintenance.


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


budd@cs.orst.edu
faugdavd@nascom.com
```

1.5 Getting Started

```
                        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,  on  the
Mac, and on the Amiga.
```

1.6 Stdin/Stdout Interface

The Stdin/Stdout Interface

Using the stdin/stdout interface, there is a prompt (the ``>'' caracter) typed to indicate the system is waiting for input. Expressions are read at the keyboard and evaluated following each carrage 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
```

1.7 Windowing Interface

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 (and the Amiga).

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 memu commands are also available. These permit you to save the current image, exit the system, or start the browser.

The browser is an interface permiting 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.

1.8 Exploring and Creating

```
                      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 passing 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  if  its  subclasses; 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 form 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 methodNamed: #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-\ 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.

1.9 New Methods and New Classes

New Methods and New Classes

With Stdin/Stdout Interface

With Windowing Interface

1.10 With Stdin/Stdout Interface

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 <- 'memacs'
```

Adding a new subclass is accomplished by sending the message "addSubClass:instanceVariableNames:" to the superclass object. The 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.

1.11 With Windowing Interface

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.

1.12 Incompatibilities

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.

1.13 Implementors Guide

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.

1.14 Finding Your Way Around

```
                   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  straightforward
     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 :
These two modules contains I/O routines.

lex.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.

1.15 Defining System Characteristics

Defining System Characteristics

There are many ways in which compilers and operating systems differ from each other. A fair amount of work has been expanded 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.

1.16 Define Options

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.

1.17 Building an Initial Image

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.
```

1.18 Object Memory

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 coverting 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.

1.19 The Bottom End

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.

1.20 Editing

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.

1.21 Editing Under Unix

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 variabled "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".

1.22 Editing on the Macintosh

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.

1.23 Input/Output Commands

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 standard 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.

1.24 Primitives

Primitives

Primitives are the means whereby actions that cannot be described directed in Smalltalk are performed. 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.

1.25 Installation

Installation Instructions

The following lists installation instructions for those systems to which Version 3 of Little Smalltalk has been ported at present. Note that installation involves the creation of two programs. The first, called ``initial'', is run once to create the initial object image (usually a filed called ``systemImage''). The second program is the Smalltalk interpreter. To run Smalltalk, both these files must be accessible. Systems that use the supplied Makefile run initial automatically; in some other systems you may need to do this manually.

If you receive the distribution on Mac or IBM disks and you want to run the system under Unix you must ``undo'' some of the changes described below.

1.26 Atari

Atari

I’ve been told (no first hand experience) that the code works on the Atari. I’ve set up a minimal description in env.h - could somebody tell me if the Atari supports prototypes, signals, or some of the other features?

You do have to make the ’rb’ changes described for the IBM PC (below), however you keep the rm instruction instead of DEL, and change the editor to whatever your system has (memacs?).

1.27 Gnu C Compiler

Gnu C Compiler

If at all possible, USE THE GNU C COMPILER. I have found the code to be much smaller (up to 1/3 smaller) and much faster (up to twice as fast). So far this has been used on the Sequent Balance system, and the Amiga.

Note that these sources support old style prototypes, as are used in Lightspeed C and Turbo C, and not the newer ANSI prototypes as are used in the gcc compiler. So do not define PROTO when using the gcc compiler

1.28 HP-UX

HP-UX

Simply say ‘‘make sysvtty’’ to make a version with the tty interface. (As of yet, I don’t have access to a system V system with an X-window interface, so I can’t test that code).

1.29 IBM PC

IBM PC / Turbo C compiler

NOTE: If you receive the sources on 5 1/2 disks containing both source and executable, the following changes have already been made to the system.

Define the symbol TURBOC at the beginning of the file env.h.

Edit the file file.st, changing the command used to delete files from rm to del (notice the space following the del):

```
    delete
            ('del ', name) unixCommand
```

In the file file.st change the mode on the command to save images from w to wb.

```
        saveImage: name
                scheduler critical: [
                        " first get rid of our own process "
                        scheduler removeProcess: scheduler currentProcess.
                File new;
                        name: name;
                        open: 'wb';
                        saveImage;
                        close ]
```

In a similar manner change the mode on the file open in the initialize method in file tty.st to use wb instead of w.

```
        initialize
                " initialize the initial object image "
                self createGlobals.
                File new;
                        name: 'systemImage';
                        open: 'wb';
                        saveImage;
                        close.
```

And also in tty.st change the editor from vi to me (or whatever your favorite editor happens to be).

```
                editor (<- 'me'.
```

Because of segmentation limits it is not possible to have an object table any larger than 6500 objects (the current default). This value is set by a define found in memory.h

```
  # define ObjectTableMax 6500
```

Compile in the compact mode (small code, large data).

1.30 Macintosh

Macintosh Lightspeed C

NOTE: If you get the distribution on 3 1/4 Mac Disks the source code changes described below have probably already been made for you.

The Mac distribution disk contains the following.

(a) A folder called ``C Sources'' that contains (naturally) all the C sources.

(b) A folder called ``ST Sources'' that contains (also naturally) all the Smalltalk sources, plus an application called ``initial'' that can be used to create or recreate the initial object image. To make changes to the image, simply edit the appropriate Smalltalk files, run initial, and move the file ``systemImage'' to the appropriate location.

(c) Two Lightspeed C projects called ``TextEdit'' and ``Stdwin'', containing code taken from Guido Van Rossums Standard Windows package.

(d) A file called ``systemImage'', which is the output of the application from part (b)

(e) An application called ``st'', which is the Smalltalk interpreter.

(f) A folder called ``misc'' that contains various different files, such as documentation and other things.

It is only necessary to recompile if you make changes to the C source. If you make changes to the Smalltalk source you only need to rerun the application called ``initial'' contained in the ``ST Sources'' folder.

If you get the sources from some other location (say off the net), you must make the following alterations. Change the mode on the file open in the saveImage command (in file.st) and in the initalize command (file stdwin.st). Define the symbol LIGHTC at the beginning of the file env.h (See instructions for the IBM PC above for a fuller explanation).

To compile you need Guido Van Rossum's Standard Windows package. Follow his instructions to create the stdwin and textedit projects (these are already on the distribution disk). To make the initial program, create a project ``initialProj'' with segments as follows. In the first segment place MacTraps. In the second segment place Stdwin. In the third place TextEdit. In the forth place the Unix library files math, stdio, storage, strings and unix. In the fifth place the sources filein.c, initial.c, interp.c, memory.c, names.c, news.c, primitives.c, unixio.c and winprims.c. In the sixth and final segment place lex.c and parser.c. To create the st program use the same structure, subsituting st.c for initial.c. You must check the ``separate STRS'' option on both projects.

Make sure when you run the initial object that all the Smalltalk sources are in the current directory; it does not complain if it can't open a file, it simply goes on. Also when you fileIn a file, the file must be in the current directory.

The Mac version uses the windowing interface. It is currently very fragile. (A few known bugs; can't restore from saved image files, output sometimes goes wrong places, output often doesn't appear until you click the mouse).

[It would be nice if clicking on an image file would start the Smalltalk application. If anybody knows how to make

Lightspeed C do this, let me know. Thanks].

1.31 Sequent Balance

Sequent Balance

Say ``make bsdtty'' to make a tty interface system.

1.32 TekTronix 4315

TekTronix 4315, Green Hills C Compiler

Say ``make bsdtty'' to make a tty interface system. Say ``make bsdx11'' to make an x-windows interface system (still somewhat buggy).

1.33 VAX / VMS

VAX / VMS

Since VMS doesn't understand Unix Makefiles, the distribution tape supplies a command file you can use. First define the symbol VMS near the begining of the file env.h, then execute the command file called vms.com. This makes a version using the tty interface. A VMS version using the X-windows interface has not been created yet.

1.34 Test Cases

Test Cases

One you have a running system; the following can be used to run the standard test cases. First load the file test.st. If you are using the windowing interface select the fileIn menu item and the file ``test.st'' (from the ST Sources folder), if you are using the tty interface use the following command

```
  File new; fileIn: 'test.st'
```

Then give the command to run all test cases.

```
  Test new all
```

Messages will be displayed as test cases are performed, and if any test cases fail.

1.35 Standard Windows

```
                     The Standard Window Package

     There is  an  experimental  windows style interface based on
Guido  Van  Rossums  standard  window  package.  This permits the
system  to  work  on  top of X-windows, as well as the Macintosh.
Information  on  Standard  Windows  can be obtained directly from
Guido   at   guido@mcvax.uucp,   or   mcvax!guido,   or  possibly
gvr@src.dec.com.   His  paper  mail  address is Guido van Rossum,
Center  for Mathematics and Computer Science, P.O. Box 4079, 1009
AB  Amsterdam,  The Netherlands.  Sources for the standard window
package  are  not  included on the Little Smalltalk distribution,
but  they  are  available  public  domain  by  ftp  from DEC SRC,
machine    gatekeeper.dec.com    (address    [128.45.9.52]).    The
subdirectory is pub/stdwin.  Contact Guido for more details.

     To make  the  projects  for  the  Macintosh  version, follow
Guidos  instructions.  For other versions, make a file stdw.o by
linking  together  all  of  Guidos  sources  for  your particular
system.  Here is a makefile for the X11 version, for example.

  #
  # X11 version of stdwins
  #
  x11 = caret.o draw.o font.o menu.o timer.o cutbuffer.o
  error.o general.o scroll.o window.o dialog.o event.o
  llevent.o system.o
  alfa = bind.o draw.o event.o keymap.o measure.o menu.o scroll.o stdwin.o syswin. ↩
     o
  gen = askfile.o  perror.o
  textedit = editwin.o textdbg.o textedit.o textlow.o textbrk.o
  tools = endian.o getopt.o glob.o monocase.o strdup.o swap.o
  x11files =

  stdw.o:
          ld -r -o stdw.o

     I emphasize this interface is very fragile.
```

1.36 Possible Changes

```
                      Possible Changes

     There are  a  couple of easy changes you may want to make at
your  site.   The default editor is vi (indicated by the value of
the  global  variable  set in the routine createGlobals in either
tty.st  or  stdwin.st);  this  can be changed to any other editor
you  like.  The system also prints the current object count prior
to  asking for commands from the user.  This can be eliminated by
removing  the  primitive  <2>   from the method initialize, class
Scheduler, file tty.st.
```

1.37 New Features

```
                            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.
```

1.38 Differences from Smalltalk-80

```
                   Differences between Little Smalltalk and the Smalltalk-
   80 system

       This section  describes  the  differences  between  the
  language  accepted  by  the  Little Smalltalk system and the
  language described in
               Smalltalk blue
               .   The  principal
  reasons for these changes are as follows:

  size  Classes which are largely unnecessary, or which  could
        be  easily  simulated  by other classes (e.g. Associa-
        tion, SortedCollection) have been  eliminated  in  the
        interest  of  keeping the size of the standard library
        as small as  possible.  Similarly,  indexed  instance
        variables  are  not  supported,  since  to do so would
        increase the size of every object in the  system,  and
        they can be easily simulated in those classes in which
        they are important (see below).

  portability
        Classes which depend upon  particular  hardware  (e.g.
        BitBlt)  are  not  included  as  part  of  the  Little
        Smalltalk system.  The basic  system  assumes  nothing
        more than ascii terminals.

  representation
        The  need  for  a  textual  representation  for  class
        descriptions required some small additions to the syn-
        tax for class methods (see
                Syntax Example
                ).   Similarly,
        the fact that classes and subclasses can be separately
        parsed, in either order, forced some  changes  in  the
        scoping rules for instance variables.

       The following sections describe these changes  in  more
  detail.

  3.1.  No Browser

       The Smalltalk-80 Programming Environment  described  in

               Smalltalk orange
```

is not included as part of the Little Smalltalk system. The Little Smalltalk system was designed to be little, easily portable, and to rely on nothing more than basic terminal capabilities. A windowing interface is now included, but has far less capability than Smalltalk-80.

3.2. Internal Representation Different

The internal representations of objects, including processes, interpreters, and bytecodes, is entirely different in the Little Smalltalk system from the Smalltalk-80 system described in
Smalltalk blue
.

3.3. Fewer Classes

Many of the classes described in
Smalltalk blue
are
not included as part of the Little Smalltalk basic system. Some of these are not necessary because of the decision not to include the editor, browser, and so on as part of the basic system. Others are omitted in the interest of keeping the standard library of classes small. A complete list of included classes for the Little Smalltalk system is given in

Class Descriptions
.

3.4. No Class Protocol

Protocol for all classes is defined as part of class Class. It is not possible to redefine class protocol as part of a class description, only instance protocol. The notion of metaclasses is not supported.

3.5. Cascades Different

The semantics of cascades has been simplified and generalized. The result of a cascaded expression is always the result of the expression to the left of the first semicolon, which is also the receiver for each subsequent continuation. Continuations can include multiple messages. A rather nonsensical, but illustrative, example is the following:

```
2 + 3 ; - 7 + 3 ; * 4
```

The result of this expression is 5 (the value yielded by 2 + 3). 5 is also the receiver for the message - 7, and that result (-2) is in turn the receiver for the message + 3. This last result is thrown away. 5 is then again used as the receiver for the message * 4, the result of which is also thrown away.

3.6. Instance Variable Name Scope

In the language described in
Smalltalk blue
, an
instance variable is known not only to the class protocol in which it is declared, but is also valid in methods defined for any subclasses of that class. In the Little Smalltalk system an instance variable can be referenced only within the protocol for the class in which it is declared.

3.7. Indexed Instance Variables

Implicitly defined indexed instance variables are not supported. In any class for which these are desired they can be easily simulated by including an additional instance variable, containing an Array, and including the following methods:

```
Class Whatever
| indexVars |
[
    new: size
        indexVars <- Array new: size

|   at: location
        ^ indexVars at: location

|   at: location put: value
        indexVars at: location put: value

    ...
```

The message new: can be used with any class, with an effect similar to new. That is, if a new instance of the class is created by sending the message new: to the class variable, the message is immediately passed on to the new instance, and the result returned is used as the result of the creation message.

3.8. No Pool Variables

The concepts of pool variables, global variables, or class variables are not supported. In their place there is a new pseudo-variable, smalltalk, which responds to the messages at: and at:put:. The keys for this collection can be arbitrary. Although this facility is available, its use is often a sign of poor program design, and should be avoided.

3.9. No Associations

The class Dictionary stores keys and values separately, rather than as instances of Association. The class Association, and all messages referring to Associations have been removed.

3.10. Generators in place of Streams

The notion of stream has been replaced by the slightly different notion of generators, in particular the use of the messages first and next in subclasses of Collection. External files are supported by an explicit class File.

3.11. Primitives Different

Both the syntax and the use of primitives has been changed. Primitives provide an interface between the Smalltalk world and the underlying system, permitting the execution of operations that cannot be specified in Smalltalk. In Little Smalltalk, primitives cannot fail and must return a value (although they may, in error situations, print an error message and return nil). The syntax for primitives has been altered to permit the specification of primitives with an arbitrary number of arguments. The format for a primitive call is as follows:

```
<primitive number argumentlist >
```

Where number is the number of the primitive to be executed (which must be a value between 1 and 255), and argumentlist is a list of Smalltalk primary expressions (see Appendix 2).

Primitive numbers
lists the meanings of each of the
currently recognized primitive numbers.

3.12. Byte Arrays

A new syntax has been created for defining an array composed entirely of unsigned integers in the range 0-255. These arrays are given a very tight encoding. The syntax is a pound sign, followed by a left square brace, followed by a sequence of numbers in the range 0 to 255, followed by a right square brace.

```
#[ numbers ]
```

Byte Arrays are used extensively internally.

3.13. New Pseudo Variables

In addition to the pseudo variable smalltalk already mentioned, another pseudo variable, selfProcess, has beed added to the Little Smalltalk system. selfProcess returns the currently executing process, which can then be passed as an argument to a semaphore, or be used as a receiver for a message valid for class Process. Like self and super, selfProcess cannot be used at the command level.

3.14. No Dependency

The notion of dependency, and automatic dependency updating, is not included in Little Smalltalk.

1.39 Class Descriptions

The messages accepted by the classes included in the Little Smalltalk standard library are described in the following pages. A list of the classes defined, where indentation is used to imply subclassing, is given below. Note that this is actually the structure from version 1 of Little Smalltalk. The proper structure for version 3 is in the running system. Try it out!

```
Object

UndefinedObject

Symbol

Boolean

True

False

Magnitude

Char

Number

Integer

Float

Radian

Point

Random

Collection

Bag

Set

KeyedCollection

Dictionary

Smalltalk

SequenceableCollection
```

```
                 Interval
                                      LinkedList

                 Semaphore

                 File

                 ArrayedCollection

                 Array

                 ByteArray

                 String

                 Block

                 Class

                 Process
```

1.40 Class Object

```
Object

     The class Object is a superclass of all classes in  the
system,  and  is  used  to  provide a consistent basic func-
tionality and  default  behavior.   Many  methods  in  class
Object are overridden in subclasses.

Responds to

  ==            Return true if receiver and argument are  the
                same object, false otherwise.

  ~~            Inverse of ==.

  asString      Return  a  string  representation  of  the
                receiver,  by  default  this  is  the same as
                printString, although one  or  the  other  is
                redefined in many subclasses.

  asSymbol      Return a symbol representing the receiver.

  class         Return object representing the class  of  the
                receiver.

  copy          Return shallowCopy of receiver.   Many  subc-
                lasses redefine shallowCopy.

  deepCopy      Return the receiver.  This  method  is  rede-
                fined in many subclasses.

  do:           The argument must be a  one  argument  block.
                Execute  the  block  on  every element of the
```

	receiver collection. Elements in the receiver collection are listed using first and next (below), so the default behavior is merely to execute the block using the receiver as argument.
error:	Argument must be a String. Print argument string as error message. Return nil.
first	Return first item in sequence, which is by default simply the receiver. See next, below.
isKindOf:	Argument must be a Class. Return true if class of receiver, or any superclass thereof, is the same as argument.
isMemberOf:	Argument must be a Class. Return true if receiver is instance of argument class.
isNil	Test whether receiver is object nil.
next	Return next item in sequence, which is by default nil. This message is redefined in classes which represent sequences, such as Array or Dictionary.
notNil	Test if receiver is not object nil.
print	Display print image of receiver on the standard output.
printString	Return a string representation of receiver. Objects which do not redefine printString, and which therefore do not have a printable representation, return their class name as a string.
respondsTo:	Argument must be a symbol. Return true if receiver will respond to the indicated message.
shallowCopy	Return the receiver. This method is redefined in many subclasses.

Examples

	Printed result
7 ~~ 7.0	True
7 asSymbol	#7
7 class	Integer
7 copy	7
7 isKindOf: Number	True
7 isMemberOf: Number	False
7 isNil	False

```
7 respondsTo: #+                        True
```

1.41 Class UndefinedObject

```
Object
  UndefinedObject
```

The pseudo variable nil is an instance (usually the only instance) of the class UndefinedObject. nil is used to represent undefined values, and is also typically returned in error situations. nil is also used as a terminator in sequences, as for example in response to the message next when there are no further elements in a sequence.

Responds to

isNil	Overrides method found in Object. Return true.
notNil	Overrides method found in Object. Return false.
printString	Return 'nil'.

Examples

```
                                        Printed result

nil isNil                               True
```

1.42 Class Symbol

```
Object
  Symbol
```

Instances of the class Symbol are created either by their literal representation, which is a pound sign followed by a string of nonspace characters (for example #aSymbol), or by the message asSymbol being passed to an object. Symbols cannot be created using new. Symbols are guaranteed to have unique representations; that is, two symbols representing the same characters will always test equal to each other. Inside of literal arrays, the leading pound signs on symbols can be eliminated, for example: #(these are symbols).

Responds to

==	Return true if the two symbols represent the same characters, false otherwise.
asString	Return a String representation of the symbol

```
             without the leading pound sign.

printString  Return a String representation of the symbol,
             including the leading pound sign.

Examples

                                        Printed result

#abc == #abc                            True
#abc == #ABC                            False
#abc ~~ #ABC                            True
#abc printString                        #abc
'abc' asSymbol                          #abc
```

1.43 Class Boolean

```
Object
  Boolean
```

The class Boolean provides protocol for manipulating true and false values. The pseudo variables true and false are instances of the subclasses of Boolean; True and False, respectively. The subclasses True and False, in combination with blocks, are used to implement conditional control structures. Note, however, that the bytecodes may optimize conditional tests by generating code in-line, rather than using message passing. Note that bit-wise boolean operations are provided by class Integer.

Responds to

```
  &            The argument must be a boolean.   Return  the
               logical conjunction (and) of the two values.

  |            The argument must be a boolean.   Return  the
               logical disjunction (or) of the two values.

  and:         The argument must be  a  block.   Return  the
               logical  conjunction (and) of the two values.
               If the receiver is false the second  argument
               is  not  used,  otherwise  the  result is the
               value  yielded  in  evaluating  the  argument
               block.

  or:          The argument must be  a  block.   Return  the
               logical  disjunction  (or) of the two values.
               If the receiver is true the  second  argument
               is  not  used,  otherwise  the  result is the
               value  yielded  in  evaluating  the  argument
               block.

  eqv:         The argument must be a boolean.   Return  the
               logical equivalence (eqv) of the two values.
  xor:         The argument must be a boolean.   Return  the
               logical exclusive or (xor) of the two values.

Examples

                                                 Printed result

(1 > 3) & (2 < 4)                                False
(1 > 3) | (2 < 4)                                True
(1 > 3) and: [2 < 4]                             False
```

1.44 Class True

```
Object
  Boolean
    True

     The pseudo variable true is an  instance  (usually  the
only instance) of the class True.

Responds To

  ifTrue:      Return the result of evaluating the  argument
               block.

  ifFalse:     Return nil.

  ifTrue:ifFalse:
               Return the result  of  evaluating  the  first
               argument block.

  ifFalse:ifTrue:
               Return the result of  evaluating  the  second
               argument block.

  not          Return false.

Examples

                                                 Printed result

(3 < 5) not                                      False
(3 < 5) ifTrue: [17]                             17
```

1.45 Class False

```
Object
  Boolean
    False

     The pseudo variable false is an instance  (usually  the
only instance) of the class False.
```

```
  ifTrue:      Return nil.

  ifFalse:     Return the result of evaluating the  argument
               block.

  ifTrue:ifFalse:
               Return the result of  evaluating  the  second
               argument block.

  ifFalse:ifTrue:
               Return the result  of  evaluating  the  first
               argument block.

  not          Return true.

Examples

                                        Printed result

(1 < 3) ifTrue: [17]                    17
(1 < 3) ifFalse: [17]                   nil
```

1.46 Class Magnitude

```
Object
  Magnitude
```

The class Magnitude provides protocol for those subclasses possessing a linear ordering. For the sake of efficiency, most subclasses redefine some or all of the relational messages. All methods are defined in terms of the basic messages <, = and >, which are in turn defined circularly in terms of each other. Thus each subclass of Magnitude must redefine at least one of these messages.

```
  <            Relational  less  than   test.   Returns   a
               boolean.

  <=           Relational less than or equal test.

  =            Relational  equal  test.  Note  that  this
               differs  from ==, which is an object equality
               test.

  ~=           Relational not equal test, opposite of =.

  >=           Relational greater than or equal test.

  >            Relational greater than test.

  between:and: Relational test for inclusion.

  max:         Return the maximum of the receiver and  argu-
               ment value.
  min:          Return the minimum of the receiver and  argu-
                ment value.

Examples

                                                    Printed result

$A max: $a                                          $a
4 between: 3.1 and: (17/3)                          True
```

1.47 Class Char

```
Object
  Magnitude
    Char

     This class defines protocol for objects with  character
values.   Characters possess an ordering given by the under-
lying representation, however arithmetic is not defined  for
character  values.   Characters  are  written  literally  by
preceding the character desired  with  a  dollar  sign,  for
example: $a   $B   $$.

Responds to

  ==            Object equality test.  Two instances  of  the
                same character always test equal.

  asciiValue    Return  an  Integer  representing  the  ascii
                value of the receiver.

  asLowercase   If  the  receiver  is  an  uppercase   letter
                returns  the same letter in lowercase, other-
                wise returns the receiver.

  asUppercase   If the receiver is a lowercase letter returns
                the   same   letter   in  uppercase,  otherwise
                returns the receiver.

  asString      Return a length  one  string  containing  the
                receiver.   Does  not  contain leading dollar
                sign, compare to printString.

  digitValue    If the  receiver  represents  a  number  (for
                example  $9)  return  the  digit value of the
                number.  If  the  receiver  is  an  uppercase
                letter  (for  example $B) return the position
                of the number in the uppercase letters +  10,
                ($B   returns   11,  for  example).   If  the
                receiver is neither a digit nor an  uppercase
                letter an error is given and nil returned.

  isAlphaNumericRespond true if receiver is either digit  or
                letter, false otherwise.
```

isDigit	Respond true if receiver is a digit, false otherwise.
isLetter	Respond true if receiver is a letter, false otherwise.
isLowercase	Respond true if receiver is a lowercase letter, false otherwise.
isSeparator	Respond true if receiver is a space, tab or newline, false otherwise.
isUppercase	Respond true if receiver is an uppercase letter, false otherwise.
isVowel	Respond true if receiver is $a, $e, $i, $o or $u, in either upper or lower case.
printString	Respond with a string representation of the character value. Includes leading dollar sign, compare to asString, which does not include $.

Examples

```
                                        Printed result

$A < $0                                 False
$A asciiValue                           65
$A asString                             A
$A printString                          $A
$A isVowel                              True
$A digitValue                           10
```

1.48 Class Number

```
Object
  Magnitude
    Number
```

The class Number is an abstract superclass for Integer and Float. Instances of Number cannot be created directly. Relational messages and many arithmetic messages are redefined in each subclass for arguments of the appropriate type. In general, an error message is given and nil returned for illegal arguments.

Responds To

+	Mixed type addition.
-	Mixed type subtraction.
*	Mixed type multiplication

/	Mixed type division.
^	Exponentiation, same as raisedTo: .
@	Construct a point with coordinates being the receiver and the argument.
abs	Absolute value of the receiver.
exp	e raised to the power.
gamma	Return the gamma function (generalized factorial) evaluated at the receiver.
ln	Natural logarithm of the receiver.
log:	Logarithm in the given base.
negated	The arithmetic inverse of the receiver.
negative	True if the receiver is negative.
pi	Return the approximate value of the receiver multiplied by (3.1415926...).
positive	True if the receiver is positive.
radians	Argument converted into radians.
raisedTo:	The receiver raised to the argument value.
reciprocal	The arithmetic reciprocal of the receiver.
roundTo:	The receiver rounded to units of the argument.
sign	Return -1, 0 or 1 depending upon whether the receiver is negative, zero or positive.
sqrt	Square root. nil if receiver is less than zero.
squared	Return the receiver multiplied by itself.
strictlyPositive	True if the receiver is greater than zero.
to:	Interval from receiver to argument value with step of 1.
to:by:	Interval from receiver to argument in given steps.
truncatedTo:	The receiver truncated to units of the argument.

```
Examples

                                        Printed result

3 < 4.1                                 True
3 + 4.1                                 7.1
3.14159 exp                             23.1406
9 gamma                                 40320
5 reciprocal                            0.2
0.5 radians                             0.5 radians
13 roundTo: 5                           15
13 truncateTo: 5                        10
```

1.49 Class Integer

```
Object
  Magnitude
    Number
      Integer

     The class Integer provides protocol  for  objects  with
integer values.

Responds To

  ==           Object   equality   test.    Two   integers
               representing the same value are considered to
               be the same object.

  //           Integer quotient, truncated towards  negative
               infinity (compare to quo:).

  \           Integer remainder, truncated towards negative
               infinity (compare to rem:).

  allMask:     Argument must be Integer.  Treating  receiver
               and  argument  as bit strings, return true if
               all bits with 1 value in argument  correspond
               to bits with 1 values in the receiver.

  anyMask:     Argument must be Integer.  Treating  receiver
               and  argument  as bit strings, return true if
               any bit with 1 value in argument  corresponds
               to a bit with 1 value in the receiver.

  asCharacter  Return the  Char  with  the  same  underlying
               ascii  representation  as the low order eight
               bits of the receiver.

  asFloat      Floating point value with same  magnitude  as
               receiver.

  bitAnd:      Argument   must   be   Integer.   Treating   the
               receiver  and argument as bit strings, return
               logical and of values.
```

bitAt:	Argument must be Integer greater than 0 and less than underlying word size. Treating receiver as a bit string, return the bit value at the given position, numbering from low order (or rightmost) position.
bitInvert	Return the receiver with all bit positions inverted.
bitOr:	Return logical or of values.
bitShift:	Treating the receiver as a bit string, shift bit values by amount indicated in argument. Negative values shift right, positive left.
bitXor:	Return logical exclusive-or of values.
even	Return true if receiver is even, false otherwise.
factorial	Return the factorial of the receiver. Return as Float for large numbers.
gcd:	Argument must be Integer. Return the greatest common divisor of the receiver and argument.
highBit	Return the location of the highest 1 bit in the receiver. Return nil for receiver zero.
lcm:	Argument must be Integer. Return least common multiple of receiver and argument.
noMask:	Argument must be Integer. Treating receiver and argument as bit strings, return true if no 1 bit in the argument corresponds to a 1 bit in the receiver.
odd	Return true if receiver is odd, false otherwise.
quo:	Return quotient of receiver divided by argument.
radix:	Return a string representation of the receiver value, printed in the base represented by the argument. Argument value must be less than 36.
rem:	Remainder after receiver is divided by argument value.
timesRepeat:	Repeat argument block the number of times given by the receiver.

Examples

```
                                        Printed result

5 + 4                                   7
5 allMask: 4                            True
4 allMask: 5                            False
5 anyMask: 4                            True
5 bitAnd: 3                             1
5 bitOr: 3                              7
5 bitInvert                             -6
254 radix: 16                           16rFE
-5 // 4                                 -2
-5 quo: 4                               -1
-5 \ 4                                 1
-5 rem: 4                               -1
8 factorial                             40320
```

1.50 Class Float

```
Object
  Magnitude
    Number
      Float
```

The class Float provides protocol for objects with floating point values.

Responds To

==	Object equality test. Return true if the receiver and argument represent the same floating point value.
^	Floating exponentiation.
arcCos	Return a Radian representing the arcCos of the receiver.
arcSin	Return a Radian representing the arcSin of the receiver.
arcTan	Return a Radian representing the arcTan of the receiver.
asFloat	Return the receiver.
ceiling	Return the Integer ceiling of the receiver.
coerce:	Coerce the argument into being type Float.
exp	Return e raised to the receiver value.
floor	Return the Integer floor of the receiver.
fractionPart	Return the fractional part of the receiver.

gamma	Return the value of the gamma function applied to the receiver value.
integerPart	Return the integer part of the receiver.
ln	Return the natural log of the receiver.
radix:	Return a string containing the printable representation of the receiver in the given radix. Argument must be an Integer less than 36.
rounded	Return the receiver rounded to the nearest integer.
sqrt	Return the square root of the receiver.
truncated	Return the receiver truncated to the nearest integer.

Examples

```
                                        Printed result

4.2 * 3                                 12.6
2.1 ^ 4                                 19.4481
2.1 raisedTo: 4                         19.4481
0.5 arcSin                              0.523599 radians
2.1 reciprocal                          0.47619
4.3 sqrt                                2.07364
```

1.51 Class Radian

```
Object
  Magnitude
    Radian
```

The class Radian is used to represent radians. Radians are a unit of measurement, independent of other numbers. Only radians will responds to the trigonometric functions such as sin and cos. Numbers can be converted into radians by passing them the message radians. Similarly, radians can be converted into numbers by sending them the message asFloat. Notice that only a limited range of arithmetic operations are permitted on Radians. Radians are normalized to be between 0 and 2PI.

Responds to

+	Argument must be a Radian. Add the two radians together and return the normalized result.
-	Argument must be a Radian. Subtract the

	argument from the receiver and return the normalized result.
*	Argument must be a Number. Multiply the receiver by the argument amount and return the normalized result.
/	Argument must be a Number. Divide the receiver by the argument amount and return the normalized result.
asFloat	Return the receiver as a floating point number.
cos	Return a floating point number representing the cosine of the receiver.
sin	Return a floating point number representing the sine of the receiver.
tan	Return a floating point number representing the tangent of the receiver.

Examples

	Printed result
0.5236 radians sin	0.5
0.5236 radians cos	0.866025
0.5236 radians tan	0.577352
0.5 arcSin asFloat	0.523599

1.52 Class Point

```
Object
  Magnitude
    Point
```

Points are used to represent pairs of quantities, such as coordinate pairs.

Responds To

<	True if both values of the receiver are less than the corresponding values in the argu- ment.
<=	True if the first value is less than or equal to the corresponding value in the argument, and the second value is less than the corresponding value in the argument.
>=	True if both values of the receiver are greater than or equal to the corresponding values in the argument.

*	Return a new point with coordinates multiplied by the argument value.
/	Return a new point with coordinates divided by the argument value.
//	Return a new point with coordinates divided by the argument value.
+	Return a new point with coordinates offset by the corresponding values in the argument.
abs	Return a new point with coordinates having the absolute value of the receiver.
dist:	Return the Euclidean distance between the receiver and the argument point.
max:	The argument must be a Point. Return the lower right corner of the rectangle defined by the receiver and the argument.
min:	The argument must be a Point. Return the upper left corner of the rectangle defined by the receiver and the argument.
transpose	Return a new point with coordinates being the transpose of the receiver.
x	Return the first coordinate of the receiver.
x:	Set the first coordinate of the receiver.
x:y:	Sets both coordinates of the receiver.
y	Return the second coordinate of the receiver.
y:	Set the second coordinate of the receiver.

```
Examples

                                        Printed result

(10@12) < (11@14)                       True
(10@12) < (11@11)                       False
(10@12) max: (11@11)                    11@12
(10@12) min: (11@11)                    10@11
(10@12) dist: (11@14)                   2.23607
(10@12) transpose                       12@10
```

1.53 Class Random

```
Object
  Random
```

The class Random provides protocol for random number generation. Sending the message next to an instance of Random results in a Float between 0.0 and 1.0, randomly distributed. By default, the pseudo random sequence is the same for each object in class Random. This can be altered using the message randomize.

Responds To

between:and:	Return a random number uniformly distributed between the two arguments.
first	Return a random number between 0.0 and 1.0. This message merely provides consistency with protocol for other sequences, such as Arrays or Intervals.
next	Return a random number between 0.0 and 1.0.
next:	Return an Array containing the next n random numbers, where n is the argument value.
randInteger:	The argument must be an integer. Return a random integer between 1 and the value given.
randomize	Change the pseudo-random number generator seed by a time dependent value.

Examples

```
                                        Printed result

i <- Random new
i next                                  0.759
i next                                  0.157
i next: 3                               #( 0.408 0.278 0.547 )
i randInteger: 12                       5
i between: 4 and: 17.5                  10.0
```

1.54 Class Collection

```
Object
  Collection
```

The class Collection provides protocol for groups of objects, such as Arrays or Sets. The different forms of collections are distinguished by several characteristics, among them whether the size of the collection is fixed or unbounded, the presence or absence of an ordering, and their insertion or access method. For example, an Array is a collection with a fixed size and ordering, indexed by integer keys. A Dictionary, on the other hand, has no fixed size or ordering, and can be indexed by arbitrary elements. Nevertheless, Arrays and Dictionarys share many features in common, such as their access method (_at: and at:put:), and the ability to respond to collect:, select:, and many other messages.

The table below lists some of the characteristics of several forms of collections:

Name	Creation Method	Size fixed?	Ordered?	Insertion method	Access method
Bag/Set	new	no	no	add:	includes:
Dictionary	new	no	no	at:put:	at:
Interval	n to: m	yes	yes	none	at:
List	new	no	yes	addFirst: addLast:	first last
Array	new:	yes	yes	at:put:	at:
String	new:	yes	yes	at:put:	at:

The list below shows messages that are shared in common by all collections.

Responds to

addAll:	The argument must be a Collection. Add all the elements of the argument collection to the receiver collection.
asArray	Return a new collection of type Array containing the elements from the receiver collection. If the receiver was ordered, the elements will be in the same order in the new collection, otherwise the elements will be in an arbitrary order.
asBag	Return a new collection of type Bag containing the elements from the receiver collection.
asList	Return a new collection of type List containing the elements from the receiver collection. If the receiver was ordered, the elements will be in the same order in the new collection, otherwise the elements will be in an arbitrary order.
asSet	Return a new collection of type Set contain-

	ing the elements from the receiver collection.
asString	Return a new collection of type String containing the elements from the receiver collection. The elements to be included must all be of type Character. If the receiver was ordered, the elements will be in the same order in the new collection, otherwise the elements will be listed in an arbitrary order.
coerce:	The argument must be a collection. Return a collection, of the same type as the receiver, containing elements from the argument collection. This message is redefined in most subclasses of collection.
collect:	The argument must be a one argument block. Return a new collection, like the receiver, containing the result of evaluating the argument block on each element of the receiver collection.
detect:	The argument must be a one argument block. Return the first element in the receiver collection for which the argument block evaluates true. Report an error and return nil if no such element exists. Note that in unordered collections (such as Bags or Dictionarys) the first element to be encountered that will satisfy the condition may not be easily predictable.
detect:ifAbsent:	Return the first element in the receiver collection for which the first argument block evaluates true. Return the result of evaluating the second argument if no such element exists.
do:	The argument must be a one argument block. Evaluate the argument block on each element in the receiver collection.
includes:	Return true if the receiver collection contains the argument.
inject:into:	The first argument must be a value, the second a two argument block. The second argument is evaluated once for each element in the receiver collection, passing as arguments the result of the previous evaluation (starting with the first argument) and the element. The value returned is the final value generated.

```
  isEmpty      Return true if the receiver  collection  con-
               tains no elements.

  occurrencesOf:Return the  number  of  times  the  argument
               occurs in the receiver collection.

  remove:      Remove the argument from the receiver collec-
               tion.   Report an error if the element is not
               contained in the receiver collection.

  remove:ifAbsent:
               Remove the first argument from  the  receiver
               collection.   Evaluate the second argument if
               not present.

  reject:      The argument must be a  one  argument  block.
               Return  a  new  collection  like the receiver
               containing all elements for which  the  argu-
               ment block returns false.

  select:      The argument must be a  one  argument  block.
               Return  a  new  collection  like the receiver
               containing all elements for which  the  argu-
               ment block returns true.

  size         Return the number of elements in the receiver
               collection.


Examples

                                        Printed result

i <- 'abacadabra'
i size                                  10
i asArray                               #( $a $b $a $c $a $d $a $b $r $a )
i asBag                                 Bag ( $a $a $a $a $a $r $b $b $c $d)
i asSet                                 Set ( $a $r $b $c $d )
i occurrencesOf: $a                     5
i reject: [:x | x isVowel]              bcdbr
```

1.55 Bag/Set

```
Object
  Collection
    Bag/Set

     Bags and Sets are each unordered  collections  of  ele-
ments. Elements in the collections do not have keys, but are
added and removed directly.  The difference  between  a  Bag
and a Set is that each element can occur any number of times
in a Bag, whereas only one copy is inserted into a Set.

Responds to
```

```
  add:         Add the indicated  element  to  the  receiver
               collection.

  add:withOccurences:
               (Bag only) Add the indicated element  to  the
               receiver Bag the given number of times.

  first        Return the first element  from  the  receiver
               collection.   As the collection is unordered,
               the first element depends upon certain values
               in  the  internal  representation, and is not
               guaranteed to be any specific element in  the
               collection.

  next         Return the next element  in  the  collection.
               In  conjunction  with first, this can be used
               to access each element of the  collection  in
               turn.

Examples

                                        Printed result

i <- (1 to: 6) asBag                    Bag ( 1 2 3 4 5 6 )
i size                                  6
i select: [:x | (x \ 2) strictlyPositive]Bag ( 1 3 5 )
i collect: [:x | x \ 3]                Bag ( 0 0 1 1 2 2 )
j <- ( i collect: [:x | x \ 3] ) asSet Set ( 0 1 2 )
j size                                  3

Note: Since Bags and Sets are unordered, there is no way  to
establish a mapping between the elements of the Bag i in the
example above and the corresponding elements in the  collec-
tion that resulted from the message collect: [:x | x \ 3].
```

1.56 KeyedCollection

```
Object
  Collection
    KeyedCollection

     The class KeyedCollection provides protocol for collec-
tions with keys, such as Dictionarys and Arrays.  Since each
entry in the collection has both a key and value, the method
add: is no longer appropriate.  Instead, the method at:put:,
which provides both a key and a value, must be used.

Responds to

  asDictionary Return a new collection  of  type  Dictionary
               containing  the  elements  from  the receiver
               collection.

  at:          Return the item in  the  receiver  collection
```

	whose key matches the argument. Produces and error message, and returns nil, if no item is currently in the receiver collection under the given key.
at:ifAbsent:	Return the element stored in the dictionary under the key given by the first argument. Return the result of evaluating the second argument if no such element exists.
atAll:put:	The first argument must be a collection containing keys valid for the receiver. At each location given by a key in the first argument place the second argument.
binaryDo:	The argument must be a two argument block. This message is similar to do:, however both the key and the element value are passed as argument to the block.
includesKey:	Return true if the indicated key is valid for the receiver collection.
indexOf:	Return the key value of the first element in the receiver collection matching the argument. Produces an error message if no such element exists. Note that, as with the message detect:, in unordered collections the first element may not be related in any way to the order in which elements were placed into the collection, but is rather implementation dependent.
indexOf:ifAbsent:	Return the key value of the first element in the receiver collection matching the argument. Return the result of evaluating the second argument if no such element exists.
keys	Return a Set containing the keys for the receiver collection.
keysDo:	The argument must be a one argument block. Similar to do:, except that the values passed to the block are the keys of the receiver collection.
keysSelect:	Similar to select, except that the selection is made on the basis of keys instead of values.
removeKey:	Remove the object with the given key from the receiver collection. Print an error message, and return nil, if no such object exists. Return the value of the deleted item.
removeKey:ifAbsent:	

```
                Remove the object with the given key from the
                receiver  collection.   Return  the result of
                evaluating the second  argument  if  no  such
                object exists.

  values        Return a Bag containing the values  from  the
                receiver collection.

Examples

                                        Printed result

i <- 'abacadabra'
i atAll: (1 to: 7 by: 2) put: $e        ebecedebra
i indexOf: $r                           9
i atAll: i keys put: $z                 zzzzzzzzzz
i keys                                  Set ( 1 2 3 4 5 6 7 8 9 10 )
i values                                Bag ( $z $z $z $z $z $z $z $z $z $z )
#(how odd) asDictionary                 Dictionary ( 1 @ #how 2 @ odd )
```

1.57 Dictionary

```
Object
  Collection
    KeyedCollection
      Dictionary

     A Dictionary is an unordered collection of elements, as
are  Bags and Sets.  However, unlike these collections, ele-
ments inserted and removed from a Dictionary must  reference
an explicit key.  Both the key and value portions of an ele-
ment can be any  object,  although  commonly  the  keys  are
instances of Symbol or Number.

Responds to

  at:put:       Place the second argument into  the  receiver
                collection  under  the key given by the first
                argument.

  currentKey    Return the key of the last element yielded in
                response to a first or next request.

  first         Return the first element of the receiver col-
                lection.   Return nil if the receiver collec-
                tion is empty.

  next          Return the next element of the receiver  col-
                lection, or nil if no such element exists.

Examples

                                        Printed result

i <- Dictionary new
```

```
i at: #abc put: #def
i at: #pqr put: #tus
i at: #xyz put: #wrt
i print                      Dictionary ( #abc @ #def #pqr @ #tus #xyz @ #wrt )
i size                       3
i at: #pqr                   #tus
i indexOf: #tus              #pqr
i keys                       Set ( #abc #pqr #xyz )
i values                     Bag ( #wrt #def # tus )
i collect: [:x | x asString at: 2]Dictionary ( #abc @ $e #pqr @ $u #xyz @ $r)
```

1.58 Smalltalk

```
Object
  Collection
    KeyedCollection
      Dictionary
        Smalltalk

     The class Smalltalk provides protocol  for  the  pseudo
variable  smalltalk.   Since it is a subclass of Dictionary,
this variable can be used to  store  information,  and  thus
provide  a  means  of  communication between objects.  Other
messages  modify  various  parameters  used  by  the  Little
Smalltalk system.

Responds To

  date         Return the current date and time as a string.

  display      Set execution display to display  the  result
               of   every  expression  typed,  but  not  for
               assignments.  Note that the display  behavior
               can also be modified using the -d argument on
               the command line.

  displayAssignSet execution display to display  the  result
               of  every expression typed, including assign-
               ment statements.

  doPrimitive:withArguments:
               Execute the indicated  primitive  with  argu-
               ments given by the second array.  A few prim-
               itives (such as those  dealing  with  process
               management)   cannot   be  executed  in  this
               manner.

  noDisplay    Turn off execution display - no results  will
               be  displayed  unless explicitly requested by
               the user.

  perform:withArguments:
               Send indicated message to the receiver, using
               the  arguments given.  The first value in the
               argument array is taken to be the receiver of
```

```
                the  message.   Unpredictable  results if the
                number of arguments is  not  appropriate  for
                the given message.

  sh:           The argument, which must be a string, is exe-
                cuted  as  a  Unix command by the shell.  The
                value returned is the termination  status  of
                the shell.

  time:         The argument must be a block.  The  block  is
                executed,  and  the number of seconds elapsed
                during  execution  returned.   Time  is  only
                accurate to within about one second.

Examples

                                        Printed result

smalltalk date                          Fri Apr 12 16:15:42 1985
smalltalk perform: #+ withArguments: #(2 5)7
smalltalk doPrimitive: 10 withArguments: #(2 5)7
```

1.59 SequenceableCollection

```
Object
  Collection
    KeyedCollection
      SequenceableCollection

     The class SequenceableCollection contains protocol  for
collections that have a definite sequential ordering and are
indexed by integer keys.  Since there is a fixed  order  for
elements,  it  is possible to refer to the last element in a
SequenceableCollection.

Responds to

  ,             Appends  the  argument  collection   to   the
                receiver  collection, returning a new collec-
                tion of the same type as the receiver.

  copyFrom:to: Return a new collection, like  the  receiver,
                containing  the  designated subportion of the
                receiver collection.

  copyWith:     Return a new collection, like  the  receiver,
                with the argument added to the end.

  copyWithout: Return a new collection, like  the  receiver,
                with all occurrences of the argument removed.

  equals:startingAt:
                The first argument must be a SequenceableCol-
                lection.   Return true if each element of the
                receiver   collection   is   equal   to   the
```

```
                 corresponding  element in the argument offset
                 by the amount given in the second argument.

    findFirst:   Find the key  for  the  first  element  whose
                 value  satisfies the argument block.  Produce
                 an error message if no such element exists.

    findFirst:ifAbsent:
                 Both arguments must be blocks.  Find the  key
                 for  the  first element whose value satisfies
                 the first argument block.  If no such element
                 exists  return  the value of the second argu-
                 ment.

    findLast:    Find the key for the last element whose value
                 satisfies  the  argument  block.   Produce an
                 error message if no such element exists.

    findLast:ifAbsent:
                 Both arguments must be blocks.  Find the  key
                 for  the  last  element whose value satisfies
                 the first argument block.  If no such element
                 exists   return   the  value  of  the  second
                 argument block.

    firstKey     Return the first key valid for  the  receiver
                 collection.

    indexOfSubCollection:startingAt:
                 Starting at the position given by the  second
                 argument,  find the next block of elements in
                 the receiver collection which match the  col-
                 lection  given  by  the  first  argument, and
                 return the index for the start of that block.
                 Produce  an error message if no such position
                 exists.

    indexOfSubCollection:startingAt:ifAbsent:
                 Similar to  indexOfSubCollection:startingAt:,
                 except that the result of the exception block
                 is produced if no  position  exists  matching
                 the pattern.

    last         Return the last element in the receiver  col-
                 lection.

    lastKey      Return the last key valid  for  the  receiver
                 collection.

    replaceFrom:to:with:
                 Replace the elements in the receiver  collec-
                 tion  in the positions indicated by the first
                 two arguments with values taken from the col-
                 lection given by the third argument.

    replaceFrom:to:with:startingAt:
                 Replace the elements in the receiver  collec-
```

	tion in the positions indicated by the first two arguments with values taken from the collection given in the third argument, starting at the position given by the fourth argument.
reversed	Return a collection, like the receiver, with elements reversed.
reverseDo:	Similar to do:, except that the items are presented in reverse order.
sort	Return a collection, like the receiver, with the elements sorted using the comparison <=. Elements must be able to respond to the binary message <=.
sort:	The argument must be a two argument block which yields a boolean. Return a collection, like the receiver, sorted using the argument to compare elements for the purpose of ordering.
with:do:	The second argument must be a two argument block. Present one element from the receiver collection and from the collection given by the first argument in turn to the second argument block. An error message is given if the collections do not have the same number of elements.

Examples

```
                                       Printed result

i <- 'abacadabra'
i copyFrom: 4 to: 8                    cadab
i copyWith: $z                         abacadabraz
i copyWithout: $a                      bcdbr
i findFirst: [:x | x > $m]             9
i indexOfSubCollection: 'dab' startingAt: 16
i reversed                             arbadacaba
i , i reversed                         abacadabraarbadacaba
i sort: [:x :y | x >= y]               rdcbbaaaaa
```

1.60 Interval

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        Interval
```

The class Interval represents a sequence of numbers in an arithmetic sequence, either ascending or descending. Instances of Interval are created by numbers in response to the message to: or to:by:. In conjunction with the message do:, Intervals create a control structure similar to do or for loops in Algol like languages. For example:

```
(1 to: 10) do: [:x | x print]
```

will print the numbers 1 through 10. Although they are a collection, Intervals cannot be added to. They can, however, be accessed randomly using the message at:.

Responds to

first	Produce the first element from the interval. In conjunction with last, this message may be used to produce each element from the interval in turn. Note that Intervals also respond to the message at:, which can be used to produce elements in an arbitrary order.
from:to:by:	Initialize the upper and lower bounds and the step size for the receiver. (This is used principally internally by the method for number to create new Intervals).
next	Produce the next element from the interval.
size	Return the number of elements that will be generated in producing the interval.

Examples

```
                                          Printed result

(7 to: 13 by: 3) asArray                  #( 7 10 13 )
(7 to: 13 by: 3) at: 2                    10
(1 to: 10) inject: 0 into: [:x :y | x + y]55
(7 to: 13) copyFrom: 2 to: 5              #( 8 9 10 11 )
(3 to: 5) copyWith: 13                    #( 3 4 5 13 )
(3 to: 5) copyWithout: 4                  #( 3 5 )
(2 to: 4) equals: (1 to: 4) startingAt: 2True
```

1.61 List

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        List
```

Lists represent collections with a fixed order, but indefinite size. No keys are used, and elements are added or removed from one end of the other. Used in this way, Lists can perform as stacks or as queues. The table below illustrates how stack and queue operations can be implemented in terms of messages to instances of List.

stack operations		queue operations	
push	addLast:	add	addLast:
pop	removeLast	first in queue	first
top	last	remove first in queue	removeFirst
test empty	isEmpty	test empty	isEmpty

Responds to

add:	Add the element to the beginning of the receiver collection. This is the same as addFirst:.
addAllFirst:	The argument must be a SequenceableCollection. The elements of the argument are added, in order, to the front of the receiver collection.
addAllLast:	The argument must be a SequenceableCollection. The elements of the argument are added, in order, to the end of the receiver collection.
addFirst:	The argument is added to the front of the receiver collection.
addLast:	The argument is added to the back of the receiver collection.
removeFirst	Remove the first element from the receiver collection, returning the removed value.
removeLast	Remove the last element from the receiver collection, returning the removed value.

Examples

```
                                        Printed result

i <- List new
i addFirst: 2 / 3                       List ( 0.6666 )
i add: $A
i addAllLast: (12 to: 14 by: 2)
i print                                 List ( 0.6666 $A 12 14 )
i first                                 0.6666
i removeLast                            14
i print                                 List ( 0.6666 $A 12 )
```

1.62 Semaphore

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        List
          Semaphore
```

Semaphores are used to synchronize concurrently running Processes.

Responds To

new:	If created using new, a Semaphore starts out with zero excess signals. Alternatively, a Semaphore can be created with an arbitrary number of excess signals by giving it an argument to new:.
signal	If there is a process blocked on the semaphore is it scheduled for execution, otherwise the number of excess signals is incremented by one.
wait	If there are excess signals associated with the semaphore the number of signals is decremented by one, otherwise the current process is placed on the semaphore queue.

1.63 File

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        File
```

A File is a type of collection where the elements of the collection are stored on an external medium, typically a disk. For this reason, although most operations on collections are defined for files, many can be quite slow in execution. A file can be opened on one of three modes: In character mode every read returns a single character from the file. In integer mode every read returns a single word, as an integer value. In string mode every read returns a single line, as a String. For writing, character and string modes will write the string representation of the argument, while integer mode must write only a single integer.

Responds To

at:	Return the object stored at the indicated position. Position is given as a character count from the start of the file.

```
  at:put:      Place the object at the indicated position in
               the  file.   Position is given as a character
               count from the start of the file.

  characterModeSet the mode of the receiver file to  charac-
               ter.

  currentKey   Return the current position in the file, as a
               character count from the start of the file.

  integerMode  Set the mode of the receiver file to integer.

  open:        Open the indicated  file  for  reading.   The
               argument must be a String.

  open:for:    The for: argument must be one of 'r', 'w'  or
               'r+'  (see  fopen(3)  in the Unix programmers
               manual).  Open  the  file  in  the  indicated
               mode.

  read         Return the next object from the file.

  size         Return the size of  the  file,  in  character
               counts.

  stringMode   Set the mode of the receiver file to string.

  write:       Write the argument into the file.
```

1.64 ArrayedCollection

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        ArrayedCollection
```

```
     The class ArrayedCollection provides protocol for  col-
lections  with  a Fixed size and integer keys.  Unlike other
collections,  which  are  created  using  the  message  new,
instances of ArrayedCollection must be created using the one
argument message new:.  The argument given with this message
must  be  a  positive  integer, representing the size of the
collection to be  created.   In  addition  to  the  protocol
shown,  many  of the methods inherited from superclasses are
redefined in this class.

Responds to

  =            The argument must also  be  an  Array.   Test
               whether  the  receiver  and the argument have
               equal elements listed in the same order.

  at:ifAbsent: Return the element stored with the given key.
               Return  the  result  of evaluating the second
```

```
                argument if the key  is  not  valid  for  the
                receiver collection.

    padTo:      Return an array like the received that is  at
                least as long as the argument value.  Returns
                the receiver if it is already longer than the
                argument.

  Examples

                                        Printed result

  'small' = 'small'                     True
  'small' = 'SMALL'                     False
  'small' asArray                       #( $s $m $a $l $l)
  'small' asArray = 'small'             True
  #(1 2 3) padTo: 5                     #(1 2 3 nil nil)
  #(1 2 3) padTo: 2                     #(1 2 3)
```

1.65 Array

```
  Object
    Collection
      KeyedCollection
        SequenceableCollection
          ArrayedCollection
            Array

       Instances of the class Array are perhaps the most  com-
  monly used data structure in Smalltalk programs.  Arrays are
  represented textually by a pound sign preceding the list  of
  array elements.

  Responds to

    at:         Return the item stored in the position  given
                by  the  argument.   An error message is pro-
                duced, and nil returned, if the  argument  is
                not a valid key.

    at:put:     Store the second  argument  in  the  position
                given  by  the first argument.  An error mes-
                sage is produced, and nil  returned,  if  the
                argument is not a valid key.

    grow:       Return a new array one  element  larger  than
                the   receiver,   with   the  argument  value
                attached to the end.  This is a slightly more
                efficient  command  than  copyWith:, although
                the effect is the same.

  Examples

                                        Printed result
```

```
i <- #(110 101 97)
i size                              3
i <- i grow: 116                    #( 110 101 97 116)
i <- i collect: [:x | x asCharacter]    #( #n #e #a #t )
i asString                          neat
```

1.66 ByteArray

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        ArrayedCollection
          ByteArray
```

A ByteArray is a special form of array in which the elements must be numbers in the range 0-255. Instances of ByteArray are given a very compact encoding, and are used extensively internally in the Little Smalltalk system. A ByteArray can be represented textually by a pound sign preceding the list of array elements surrounded by a pair of square braces.

Responds to

at:	Return the item stored in the position given by the argument. An error message is produced, and nil returned, if the argument is not a valid key.
at:put:	Store the second argument in the position given by the first argument. An error message is produced, and nil returned, if the argument is not a valid key.

Examples

```
                                        Printed result

i <- #[110 101 97]
i size                                  3
i <- i copyWith: 116                    #[ 110 101 97 116 ]
i <- i asArray collect: [:x | x asCharacter]#( #n #e #a #t )
i asString                              neat
```

1.67 String

```
Object
  Collection
    KeyedCollection
      SequenceableCollection
        ArrayedCollection
```

String

Instances of the class String are similar to Arrays, except that the individual elements must be Character. Strings are represented literally by placing single quote marks around the characters making up the string. Strings also differ from Arrays in that Strings possess an ordering, given by the underlying ascii sequence.

Responds to

,	Concatenates the argument to the receiver string, producing a new string. If the argument is not a String it is first converted using printString.
<	The argument must be a String. Test if the receiver is lexically less than the argument. For the purposes of comparison case differences are ignored.
<=	Test if the receiver is lexically less than or equal to the argument.
>=	Test if the receiver is lexically greater than or equal to the argument.
>	Test if the receiver is lexically greater than the argument.
asSymbol	Return a Symbol with characters given by the receiver string.
at:	Return the character stored at the position given by the argument. Produce and error message, and return nil, if the argument does not represent a valid key.
at:put:	Store the character given by second argument at the location given by the first argument. Produce an error message, and return nil, if either argument is invalid.
copyFrom:length:	Return a substring of the receiver. The substring is taken from the indicated starting position in the receiver and extends for the given length. Produce an error message, and return nil, if the given positions are not legal.
copyFrom:to:	Return a substring of the receiver. The substring is taken from the indicated positions. Produce an error message, and return nil, if the given positions are not legal.
printAt:	The argument must be a Point which describes

```
                a  location  on  the  terminal  screen.   The
                string is printed at the specified location.

  size          Return the number of characters stored in the
                string.

  sameAs:       Return true  if  the  receiver  and  argument
                string  match  with  the  exception  of  case
                differences.  Note that the boolean message =
                ,  inherited  from  ArrayedCollection, can be
                used to see  if  two  strings  are  the  same
                including case differences.

Examples

                                        Printed result

'example' at: 2                         $x
'bead' at: 1 put: $r                    read
'small' > 'BIG'                         True
'small' sameAs: 'SMALL'                 True
'tary' sort                             arty
'Rats live on no evil Star' reversed    ratS live on no evil staR
```

1.68 Block

```
Object
  Block

     Although it is easy for  the  programmer  to  think  of
blocks  as  a  syntactic  construct, or a control structure,
they are actually objects, and share attributes of all other
objects  in  the  Smalltalk  system,  such as the ability to
respond to messages.

Responds to

  fork          Start the block executing as a Process.   The
                value  nil  is  immediately returned, and the
                Process created from the block  is  scheduled
                to run in parallel with the current process.

  forkWith:     Similar to fork, except  that  the  array  is
                passed  as  arguments  to  the receiver block
                prior to scheduling for execution.

  newProcess    A new Process is created for the  block,  but
                is not scheduled for execution.

  newProcessWith:
                Similar to newProcess, except that the  array
                is  passed as arguments to the receiver block
                prior to it being made into a process.

  value         Evaluates the receiver  block.   Produces  an
```

```
                error   message,  and  returns  nil,  if  the
                receiver block  required  arguments.   Return
                the value yielded by the block.

  value:        Evaluates the receiver  block.   Produces  an
                error   message,  and  returns  nil,  if  the
                receiver block did not require a single argu-
                ment.  Return the value yielded by the block.

  value:value: Two argument block evaluation.

  value:value:value:
                Three argument block evaluation.

  value:value:value:value:
                Four argument block evaluation.

  value:value:value:value:value:
                Five argument block evaluation.

  whileTrue:    The receiver block is  repeatedly  evaluated.
                While  it  evaluates  to  true,  the argument
                block is also evaluated.  Return nil when the
                receiver block no longer evaluates to true.

  whileTrue     The receiver block  is  repeatedly  evaluated
                until it returns a value that is not true.

  whileFalse:   The receiver block is  repeatedly  evaluated.
                While  it  evaluates  to  false, the argument
                block is also evaluated.  Return nil when the
                receiver block no longer evaluates to false.

  whileFalse    The receiver block  is  repeatedly  evaluated
                until it returns a value that is not false.

Examples

                                        Printed result

['block indeed'] value                  block indeed
[:x :y | x + y + 3] value: 5 value: 7   15
```

1.69 Class

```
Object
  Class

     The class  Class  provides  protocol  for  manipulating
class  instances.   An  instance of class Class is generated
for each class in the Smalltalk system.   New  instances  of
this  class are then formed by sending messages to the class
instance.

Responds to
```

deepCopy:	The argument must be an instance of the receiver class. A deepCopy of the argument is returned.
edit	The user is placed into a editor editing the file from which the class description was originally obtained. When the editor terminates, the class description will be reparsed and will override the previous description. See also view, listedit, and listview (below).
list	Lists all subclasses of the given class recursively. In particular, Object list will list the names of all the classes in the system.
listedit	Similar to list (above) but displays the results in a scrollable GUI window. Clicking on an entry in the list then invokes edit (above) on the selected entry.
listview	Similar to listedit (above) but invokes view (below) for the selected entry, rather than edit.
new	A new instance of the receiver class is returned. If the methods for the receiver contain protocol for new, the new instance will first be passed this message.
new:	A new instance of the receiver class is returned. If the methods for the receiver contain protocol for new:, the new instance will first be passed this message.
respondsTo	List all the messages that the current class will respond to.
respondsTo:	The argument must be a Symbol. Return true if the receiver class, or any of its superclasses, contains a method for the indicated message. Return false otherwise.
shallowCopy:	The argument must be an instance of the receiver class. A shallowCopy of the argument is returned.
superClass	Return the superclass of the receiver class.
variables	Return an array containing the names of the instance variables used in the receiver class.
view	Place the user into an editor viewing the class description from which the class was

```
                created.  Changes made to the file will  not,
                however, affect the current class representa-
                tion.

Examples

                                        Printed result

Array new: 3                            #( nil nil nil )
Bag respondsTo: #add:                   True
SequenceableCollection superClass       KeyedCollection
```

1.70 Process

```
Object
  Process

     Processes are created by the system, or by passing  the
message  newProcess  or  fork  to  a  block;  they cannot be
created directly by the user.

Responds To

  block         The  receiver  process  is  marked  as  being
                blocked.   This  is  usually  the result of a
                semaphore wait.  Blocked  processes  are  not
                executed.

  resume        If the receiver process has  been  suspended,
                it is rescheduled for execution.

  suspend       If the receiver process is scheduled for exe-
                cution, it is marked as suspended.  Suspended
                processes are not executed.

  state         The current state of the receiver process  is
                returned as a Symbol.

  terminate     The receiver process is terminated.  Unlike a
                blocked  or  suspended  process, a terminated
                process cannot be restarted.

  unblock       If the receiver process is currently blocked,
                it is scheduled for execution.

  yield         Returns nil.  As a side effect,  however,  if
                there  are pending processes the current pro-
                cess is placed back on the process queue  and
                another process started.
```

1.71 Syntax Example

```
                       The complete syntax accepted by Little Smalltalk is ↩
                          described in

                A Little Smalltalk
                , as amended by the

                incompatibilities
                 introduced by version 3.
  Look in the *.st modules for examples.
```

1.72 References

References

Budd, T. [1987] A Little Smalltalk. Reading, Mass.
Addison-Wesley
(A Little Smalltalk)

Goldberg, A. and Robson, D. [1983] Smalltalk-80: The Language
and Its Implementation. Reading, Mass. Addison-Wesley.
(Smalltalk blue)

Goldberg, A. [1983] Smalltalk-80: The Interactive
Programming Environment. Reading, Mass. Addison-Wesley.
(Smalltalk orange)