GNU Smalltalk User's Guide

GNU Smalltalk Version 1.1.1

by Steven B. Byrne

Introduction

GNU Smalltalk is an implementation that closely follows the Smalltalk-80 language (tm ParcPlace Systems) as described in the book *Smalltalk-80: the Language and its Implementation* by Adele Goldberg and David Robson, which will hereinafter be referred to as "the Blue Book".

The Smalltalk programming language is an object oriented programming language. This means, for one thing, that when programming you are thinking of not only the data that an object contains, but also of the operations available on that object. The object's data representation capabilities and the operations available on the object are "inseparable"; the set of things that you can do with an object is defined precisely by the set of operations, which Smalltalk calls *methods*, that are available for that object. You cannot even examine the contents of an object from the outside. To an outsider, the object is a black box that has some state and some operations available, but that's all you know.

In the Smalltalk language, everything is an object. This includes variables, executable procedures (methods), stack frames (called method contexts or block contexts), etc. Each object is an *instance* of a *class*. A class can be thought of as a datatype and the set of functions that operate on that datatype. An instance is a particular variable of that datatype.

When you want to perform an operation on an object, you send it a *message*, and the object performs an operation that corresponds to that message. For example, to print an object, you'd send it the message `print`, thus:

```
randomObject print
```

The message that you send is actually the name of a method (procedure) to invoke. When you send a message to an object, Smalltalk tries to find a method that's defined for that type of object. It first looks in the object's class for a method that matches. If none is found there, it looks in the object's parent class, then the grandparent class, and so on. At the top of the class structure is a class called Object, which has no parent. If the method is not found by the time that the searching gets to the methods of class Object, an error occurs. This error is signaled by sending the original object a `doesNotUnderstand:` message, which, if not intercepted by the object's class or any parent class, will be handled by Object itself by reporting the error to the user and printing a backtrace of the methods that had been invoked at the time the error occurred.

1 Installation

1.1 Which files to examine before compiling

Before compiling GNU Smalltalk, you'll want to examine some files, and adjust them to suit the environment that you're running in.

The files that need to be examined are:

1. `mstpaths.h-dist`
2. `Makefile`

`mstpaths.h-dist` needs to be copied to `mstpaths.h` and edited to reflect your directory structure. This defines where GNU Smalltalk will look by default for the kernel method definition files and where it will look for the saved Smalltalk binary image. As distributed, these default to `/usr/local/smalltalk/`. If you plan on installing the GNU Smalltalk system elsewhere, you will want to edit these in `mstpaths.h`.

Also, you may want to examine the first part of `Makefile`. This file defines which compiler you will use, the mail path to the GNU Smalltalk maintainer, etc.

1.2 Compiling GNU Smalltalk

If you have made the modifications as described in the previous section, you should be ready to build GNU Smalltalk. First, you should configure GNU Smalltalk for the particular machine and operating system that you are on. The current list of supported (i.e., known working) platforms is given in the sections on supported implementations (see Section 1.3 [Implementations], page 4).

To perform this configuration, merely type:

```
config.mst platform
```

where *platform* is one of the supported hardware/software platforms of GNU Smalltalk. This will create a file called `mstconfig.h` that is used by the GNU Smalltalk system to tailor certain behaviors.

After you've configured GNU Smalltalk, and you're satisfied with the settings in `Makefile` and `mstpaths.h`, you can compile the system by typing:

```
make
```

Smalltalk should compile and link with no errors. Before you make any further changes to the Smalltalk system, you should also do

```
make setup
```

which will create a copy of all of the Smalltalk files in a subdirectory called `./orig`. This step is optional, but useful for the following reason: any changes or fixes that you make to the GNU Smalltalk system (such as bug fixes or enhancements) can, at some later point in time, be automatically diffed by doing

```
make diffs
```

This is the preferred way to report changes or fixes to the system.

After doing a `make diffs`, and sending off the differences (or at least squirreling away a copy of the `mst.diffs` file that `make diffs` creates) you can do another `make setup` to set

the original state to be the current state. As a shortcut, you can produce your differences and mail them to the GNU Smalltalk maintainer (me :-) in one operation by doing:

```
make mail-diffs
```

Note: If you add files to the top level Smalltalk directory that you want to be included in the diffs, you must add them to the list of files in `mstfiles`. This includes top level directories. The `mstfiles` file drives the diffs system and the `make setup` system.

After you have successfully built the GNU Smalltalk interpreter, you can test it by invoking it. You should be able to invoke Smalltalk, thus:

```
mst -V
```

You should see the various classes being loaded one by one. At the end you should see a message of the form:

```
Processing CFuncs.st
Processing Autoload.st
"GC flipping to space 1...copied space = 100.0%...done"
Smalltalk 1.1.1 Ready

st>
```

At this point, you have a working GNU Smalltalk. *Congratulations!!!*

If more people than just yourself are going to be working on GNU Smalltalk, or if you just wish to be a bit more tidy, you'll probably want to put copies of the kernel method definitions in the directory that is mentioned in `mstpaths.h` as `KERNEL_PATH`. You can do this by typing (using `/usr/local/smalltalk` as an example):

```
cp *.st /usr/local/smalltalk
```

assuming that you have previously created that directory.

You will also want to create the image file in the proper place (see Section 3.1 [General Features], page 11, for info about what an image file is). You should switch to the directory that you want the Smalltalk binary image to live in, and invoke Smalltalk. It should reload the kernel definition files and produce a new binary image. One small issue with this approach is that the file names associated with the method definitions may be incorrect if the kernel definition files are in the same directory that Smalltalk is invoked in when you create the binary image. Smalltalk will function completely normally, except that if you try to get the source code for a method, you will probably lose. To avoid such lossage, you can switch to a directory that does not have the Smalltalk kernel definition files in it, run Smalltalk to create the binary image, and then move or copy the image file to its final home. Yes, this is clumsy, and yes, I will fix this in a future release.

1.3 Implementations

GNU Smalltalk is known to have run on the following machines, operating systems, and compilers:

Machine	Operating system	C compiler
Apollo 3000,4000,10000	Domain/OS 10.1	cc
Atari	TOS	gcc

```
DECStation 3100         Ultrix (2.1)            cc
Encore Multimax         ???                     cc
HP 9000/{300,800}       HP-UX (800=7.0,300=6.5) cc
Interactive 386         System V.3
Sony News 1810          NEWOS 3.2               cc
Sun3,Sun4               SunOS 3.5, 4.0.1        gcc 1.35, 1.37; cc
NeXT                    1.0                     cc (gcc 1.34)
Pyramid                 OSx4.1                  cc
Sequent                 ???                     cc
SGI Iris-4D             ???                     cc
Tektronix 431[5-7], 9?  ???                     cc
VAX                     BSD 4.3/Mach            cc
```

The names of the supported configurations (as given to `config.mst`) are:

```
config.mst target             Machine

apollo                  Apollo 3000 & 4000 (Domain/OS 10.1 or later)
apollo-88k              Apollo 10000
atari                   Atari ST
ds3100                  DECStation 3100 (2.1)
encore                  Encore Multimax
hp9k300                 HP 9000 series 300 (hp-ux 6.5)
hp9k800                 HP 9000 series 800 (hp-ux 7.0)
iris4d                  SGI Iris 4D
i386-sysv               Interactive Systems 386 System V.3
news                    Sony News 3.2
next                    NeXT, OS version 1.0
pyr-bsd                 Pyramid
sequent                 Sequent
sun-os3                 Sun 2's 3's and 4's, SunOS 3.x
sun-os4                 Sun 2's 3's and 4's, SunOS 4.0.x
tek4310                 Tektronix 431x
vax                     BSD derivative
```

1.4 Readline interface for GNU Smalltalk

The readline library is a piece of technology that allows for Emacs style command editing (control keys move forward and back through what has been typed, *C-p* and *C-n* move to previous and next commands that have been typed, etc.) from within GNU Smalltalk. It actually is the exact same readline library that comes with Bash; for that reason, it is not supplied with GNU Smalltalk.

To use readline with GNU Smalltalk, you must first obtain a version of the readline library. The readline library is now available as a standalone distribution from standard GNU archives, or you may use the one which comes with Bash distributions. Copy (or make a link to) the readline to a directory in the Smalltalk top level directory called `./readline`. Edit the `Makefile` and uncomment the line

```
#READLINE = -DUSE_READLINE
```

Now do:

```
make clean
```

This will ensure that the proper files which depend on the readline library are recompiled.

Now invoke `make`. Smalltalk should link and run normally, with the exception that the readline functionality is enabled.

2 Invoking GNU Smalltalk

2.1 Command line arguments

GNU Smalltalk may be invoked via the following command:

```
mst [ flags ... ] [ file ... ]
```

When you first invoke GNU Smalltalk, it will attempt to see if any of the kernel method definition files are newer than the last saved binary image in the current directory (if there is one). If there is a newer kernel method definition file, or if the binary image file (called `mst.im`) does not exist, a new binary image will be built by loading in all the kernel method definition files, performing a full garbage collection in order to compact the space in use, and then saving the resulting data in a position independent format. Your first invocation should look something like this:

```
"GC flipping to space 1...copied space = 100.0%...done"
Smalltalk 1.1.1 Ready


st>
```

If you specify *file*, that file will be read and executed and Smalltalk will exit when end of file is reached. If you specify more than one file, each will be read and processed in turn. If you don't specify *file*, standard input is read, and if the standard input is a terminal, a prompt is issued. You may specify `-` or `--` for the name of a file to invoke an explicit read from standard input.

The flags may be specified one at a time, or in a group. A flag or a group of flags always starts off with a dash to indicate that what follows is a flag or set of flags instead of a file name. In the current implementation the flags can be intermixed with file names, but their effect is as if they were all specified first. The various flags are interpreted as follows:

`-c` When this flag is set and a fatal signal occurs, a core dump is produced after an error message is printed and the stack has been backtraced. Normally, the backtrace is produced and the system terminates without dumping core.

`-d` Declaration tracing...prints the class name, the method name, and the byte codes that the compiler is generating as it compiles methods. Only for files that are named explicitly on the command line; kernel files that are loaded automatically as part of rebuilding the image file do not have their declarations traced.

`-D` Like the -d flag, but also includes declarations processed for the kernel files.

`-e` Prints the byte codes being executed as the interpreter operates. Only works for those executions that occur after the kernel files have been loaded and the image file dumped.

`-E` Like the -e flag, but includes all byte codes executed, whether they occur during the loading of the kernel method definition files, or during the loading and execution of user files.

`-h or -H` Prints out a brief summary of the command line syntax of GNU Smalltalk, including the definitions of all of the option flags, and then exits.

`-i` Ignore the saved image file; always load from the kernel method definition files. Setting this flag bypasses the normal checks for kernel files newer than the image file, or the image file's version stamp out of date with respect to the Smalltalk version. After the kernel definitions have been loaded, a new image file will be saved.

`-I` *file* Use the image file named *file* as the image file to load. Completely bypasses checking the file dates on the kernel files and standard image file.

`-p` This flag is typically not used by the user; it is for the Smalltalk interactor mode within GNU Emacs (see `st.el`).

`-q` Suppress the printing of execution information while GNU Smalltalk runs. Messages about the beginning of execution or how many byte codes were executed are completely suppressed when this flag is set.

`-r` Disables certain informative I/O; this is used by the regression testing system and is probably not of interest to the general user.

`-v` Prints out the Smalltalk version number. Has no other effect on execution.

`-V` Enables verbose mode. When verbose mode is on, various diagnostic messages are printed, such as the name of each file as it's loaded.

`-y` Turns on parser debugging. Not typically used.

2.2 Startup sequence

When GNU Smalltalk is invoked, it tries to find the saved binary image file. If this is found, it compares the write dates of all of the kernel method definition files against the write date of the binary image file. If any of the kernel files are newer, or if the image file cannot be found, or if the `-i` flag is set, the image file is ignored, all of the kernel method definition files are loaded and then the binary image file is saved.

Smalltalk first looks for the saved image file, called `mst.im` in the current directory, to allow for overriding the system default image file. If that doesn't exist, it checks the `SMALLTALK_IMAGE` environment variable, and if that's defined, it tries to find the image file in the directory given by `SMALLTALK_IMAGE`, which again allows individual users to override the system default location for the image file. If `SMALLTALK_IMAGE` is not defined, Smalltalk will use the definition of `IMAGE_PATH`, as defined in `mstpaths.h`, which is compiled in when the Smalltalk system is first built.

In a similar fashion, Smalltalk looks for each of the kernel method definition files in the current directory, to allow for explicit overriding of the installed method definition files. If they cannot be found in the current directory, and the `SMALLTALK_KERNEL` environment variable is defined, the directory that environment variable refers to is examined for the kernel file(s), which again allows individual users to override the system default location for kernel files. If `SMALLTALK_KERNEL` is not defined, Smalltalk uses the definition of `KERNEL_PATH`, which is also defined in `mstpaths.h` and compiled into the system.

Even if the image file is more recent than all the kernel definition files, if the version of Smalltalk that created the image file is different from the one that's trying to load it, or if the size of the OOP table is different between image save time and image load time, or if the `-I` flag is specified, the image file will be ignored.

The set of files that make up the kernel method definitions can be found in `mstmain.c`, in the *standardFiles* variable. Each file is loaded in turn. Once they have all been loaded, a full garbage collection is performed, and the entire contents of the object table and object memory are dumped to a file called `mst.im` in the current directory.

At this point, independent of whether the binary image file was loaded or created, any blocks that were marked as init blocks (see Section 3.2.5 [Init Blocks], page 15) are invoked (in the order of their declaration).

After the init blocks have been executed, the user initialization file (see Section 2.5 [Init file], page 10) (if any) is loaded.

Finally, if there were any files specified on the command line, they are loaded, otherwise standard input is read and executed until an EOF is detected.

2.3 Syntax of GNU Smalltalk

The language that Smalltalk accepts is based on the *file out* syntax as shown in the *Green Book*, also known as *Smalltalk-80: Bits of History, Words of Advice* by Glenn Krasner. The entire grammar of GNU Smalltalk is described in the mst.y file, but a brief description may be in order:

```
<statements> !
```

Executes the given statements immediately. For example,

```
16rFFFF printNl !
```

prints out the decimal value of hex `FFFF`, followed by a newline.

```
Smalltalk quitPrimitive !
```

exits from the system. You can also type a *C-d* to exit from Smalltalk if it's reading statements from standard input.

```
! <class expression> methodsFor: <category name> !
<method definition 1> !
<method definition 2> !
...
<method definition n> ! !
```

This syntax is used to define new methods in a given class. The <class expression> is an expression that evaluates to a class object, which is typically just the name of a class, although it can be the name of a class followed by the word `class`, which causes the method definitions that follow to apply to the named class itself, rather than to its instances. Two consecutive bangs terminate the set of method definitions. <category name> should be a string object that describes what category to file the methods in.

```
!Float methodsFor: 'pi calculations'!

radiusToArea
    ^self squared * Float pi !

radiusToCircumference
    ^self * 2 * Float pi ! !
```

It also bears mentioning that there are two assignment operators: `_` and `:=`. Both are usable interchangeably, provided that they are surrounded by spaces. The GNU Smalltalk

kernel code uses the `_` form exclusively, as this is the correct mapping between the assignment operator mentioned in the Blue Book and the current ASCII definition. In the ancient days (like the middle 70's), the ASCII underscore character was also printed as a back-arrow, and many terminals would display it that way, thus its current usage.

The return operator, which is represented in the Blue Book as an up-arrow, is mapped to the ASCII caret symbol `^`.

A complete treatment of the syntax of the language is beyond the scope of this document. Please refer to the Blue Book (or the Purple Book, if that's the only Smalltalk-80 book available) for details of the syntax and semantics of the Smalltalk language.

2.4 Operating GNU Smalltalk

You operate GNU Smalltalk by typing in expressions to the '`st>`' prompt, and/or reading in files that contain Smalltalk code.

At some time, you may wish to abort what GNU Smalltalk is doing and return to the command prompt. You can use `C-c` to do this. Note that the `C-c` handling is relatively new, and somewhat immature. For example, typing `C-c` while loading a file may not work, and it won't break out of C code that hasn't been called via the C callout mechanism. Still, it's a vast improvement over not having anything at all :-).

2.5 Per-user init files

When GNU Smalltalk is invoked, it will examine your home directory for a file with the name `.stinit`. If this file exists, it is loaded as a normal Smalltalk file. This file can be used for per-user customizations (such as turning off garbage collection messages), and definitions.

This file is always loaded; there is no current way to have a file loaded only before main binary image is created. In version 1.2, this problem is fixed.

2.6 Running the test suite

GNU Smalltalk comes with a set of files that provides a simple regression test suite.

To run the test suite, you should be connected to top-level the Smalltalk directory. Type

```
make regress
```

You should see the names of the test suite files as they are processed, but that's it. Any other output indicates some problem. The only system that I know of which currently fails the test suite is the NeXT, and this is apparently due to their non-standard C runtime libraries.

The test suite is by no means exhaustive. One good way to help the GNU Smalltalk project, and learn some Smalltalk in the process, is to add files and tests to the test suite directory. Ideally, the test suite would be used as the "go/nogo" gauge for whether a particular port of GNU Smalltalk is really working properly.

3 Features of GNU Smalltalk

The following sections describe of the various features of GNU Smalltalk, version 1.1.1, and discuss differences from the Smalltalk-80 language which is described in the Blue Book.

3.1 General Features

GNU Smalltalk supports the following features of general interest:

'`save binary file`'
: This allows you to snapshot the current state of the GNU Smalltalk virtual machine. This means that all objects are saved to a file that can be loaded rapidly at a later point in time. There are two messages that you can use:

  `SystemDictionary snapshot`
  : This saves the current state of the GNU Smalltalk system to a file called `mst.im`. When you later invoke GNU Smalltalk, this file will be used for binary loading if none of the kernel files has a more recent file modification date, if the GNU Smalltalk version number is the same as the one that created the image file, and if the size of the object table has not changed between the time that the image file was dumped and the time that the image file is loaded. Note that the instance's class that you send this message to is `SystemDictionary`; in practice, you send this message to the sole instance of `SystemDictionary`, `Smalltalk`.

  `SystemDictionary snapshot: aString`
  : This operates exactly like the `snapshot` message above, except that the string argument you supply is used as the name of the file to save the image to. However, since you currently cannot specify any other file name for the image file to load from other than `mst.im`, this capability is of limited utility (you could save several snapshots throughout a run, and move the one that you were happy with to `mst.im`).

'`incremental garbage collection`'
: This improves apparent performance by eliminating pauses while garbage collection runs. There is a message that is printed when a garbage collector flip occurs; this is can be suppressed by doing:

```
Smalltalk gcMessage: false!
```

'`Smalltalk editing mode for GNU Emacs`'
: This mode supports editing Smalltalk method definitions (see Section 3.2.6 [Edit], page 15, for details on how to access this package, and see Chapter 4 [Emacs], page 21, for a description of the Smalltalk editing mode for GNU Emacs).

3.2 Additional features of GNU Smalltalk

In this section, the features which are specific to GNU Smalltalk are described. These features include support for calling C functions from within Smalltalk, accessing UNIX

environment variables, and controlling various aspects of compilation and execution monitoring.

3.2.1 Using the C callout mechanism

To use the C callout mechanism, you first need to inform Smalltalk about the C functions that you wish to call. You currently need to do this in two places: 1) you need to establish the mapping between your C function's address and the name that you wish to refer to it by, and 2) define that function along with how the argument objects should be mapped to C data types to the Smalltalk interpreter. As an example, let us use the pre-defined (to GNU Smalltalk) functions of `system` and `getenv`.

First, the mapping between these functions and string names for the functions needed to be established in `mstcint.c`. In the function `initCFuncs`, the following code appears:

```
extern int system();
extern char *getenv();

defineCFunc("system", system);
defineCFunc("getenv", getenv);
```

Any functions that you will call from Smalltalk must be similarly defined in this routine.

Second, we need to define a method that will invoke these C functions and describe its arguments to the Smalltalk runtime system. Here are the definitions for the two functions `system` and `getenv` (taken from `CFuncs.st`)

```
Behavior defineCFunc: 'system'
         withSelectorArgs: 'system: aString'
         forClass: SystemDictionary
         returning: #int
         args: #(string)!

Behavior defineCFunc: 'getenv'
         withSelectorArgs: 'getenv: aString'
         forClass: SystemDictionary
         returning: #string
         args: #(string)!
```

The various keyword arguments are described below. Note that we send the defineCFunc:... message to the Behavior class to define the new methods, even though the methods may be installed in some class other than Behavior.

The arguments are as follows:

`defineCFunc: 'system'`
: This says that we are defining the C function `system`. This name must be EXACTLY the same as the string passed to the `defineCFunc` routine in initCFuncs.

`withSelectorArgs: 'system: aString'`
: This defines how this method will be invoked from Smalltalk. The name of the method does not have to match the name of the C function; we could have just as easily defined the selector to be `'rambo: fooFoo'`; it's just good practice to

define the method with a similar name and the argument names to reflect the data types that should be passed.

`forClass: SystemDictionary`
: This specifies where the new method should be stored. In our case, the method will be installed in the SystemDictionary, so that we would invoke it thus:

```
Smalltalk system: 'lpr README' !
```

Again, there is no special significance to which class receives the method; it could have just as well been Float, but it might look kind of strange to see:

```
1701.0 system: 'mail sbb@eng.sun.com' !
```

`returning: #int`
: This defines the C data type that will be returned. It is converted to the corresponding Smalltalk data type. The set of legal return types is:

```
char        Single C character value
string      A C char * value, converted to a Smalltalk string
symbol      A C char * value, converted to a Smalltalk symbol
int         A C int value
long        A C long value
double      A C double value, converted to an instance of Float
void        No returned value
cObject     An anonymous C pointer value; useful to pass back to
            some C function later
smalltalk   An anonymous (to C) Smalltalk data value; should have
            been passed to C at some point in the past.
```

`args: #(string)`
: This is an array of symbols that describes the types of the arguments in order. For example, to specify a to call open(2), the arguments might look something like:

```
args: #(string int)
```

The following argument types are supported; see above for details.

```
unknown     Smalltalk will make the best conversion that it can
            for this object; see the mapping table below
char        passed as "char", which is promoted to "int"
string      passed as "char *"
stringOut   passed as "char *", the contents are expected to be
            overwritten with a new C string, and the object that
            was passed becomes the new string on return
symbol      passed as "char *"
byteArray   passed as "char *", even though may contain NUL's
int         passed as "int"
long        passed as "long"
double      passed as "double"
variadic    an Array is expected, each of the elements of the
            array will be converted
cObject     C object value passed as "long" or "void *"
```

`smalltalk` Pass the object pointer to C. The C routine should treat the value as a pointer to anonymous storage. This pointer can be returned to Smalltalk at some later point in time.

Table of parameter conversions:

Declared param type	Object type	C parameter type used
smalltalk	Object	void *
long	Integer	long
unknown	Integer	long
int	Integer	int
char	Integer	int
int	Boolean (True, False)	int
char	Boolean (True, False)	int
unknown	Boolean (True, False)	int
long	Boolean (True, False)	long
char	Character	int (C promotion rule)
unknown	Character	int
string	String	char *
stringOut	String	char *
unknown	String	char *
symbol	Symbol	char *
string	Symbol	char *
unknown	Symbol	char *
byteArray	ByteArray	char *
unknown	ByteArray	char *
double	Float	double (C promotion)
unknown	Float	double
cObject	CObject	void *
unknown	CObject	void *
variadic	Array	each element is passed according to "unknown"

3.2.2 UNIX file-IO primitive messages

The following methods are defined for the `FileStream` class:

`open: fileName mode: fileMode`
: Returns an instance of `FileStream`. The instance accesses a file called *fileName* (which should be a `String`) with mode *fileMode* (which should also be a `String`, something that fopen(3) would accept as the mode, such as `'r'`, `'w'`, etc.). The elements of the stream are characters.

The following methods are defined for instances of `FileStream`:

`close` Close the given `FileStream`. Any further messages sent to the file stream are in error and have undefined behavior.

`next` Returns the next character from the stream (an instance of `Character`).

`nextPut: aChar`
: Stores the character (which should be an instance of `Character`) as the next character of the receiver.

`position: bytePosition`
: Sets the current position of the receiver to be *bytePosition*, with the beginning of file being denoted by zero.

`position` Returns an integer that indicates the current position within the `FileStream`. This is zero-based.

`contents` Returns a `String` that represents the entire remaining contents of the `FileStream`.

`atEnd` Returns `true` if the `FileStream` object is at end of file, false otherwise.

`size` Returns the size in bytes of the given `FileStream` instance (if known).

In addition, the three files, `stdin`, `stdout`, and `stderr` are declared as global instances of `FileStream` that are bound to the proper values.

Also, `Object` defines two other methods: `print` and `printNl`. These both do a `printOn:` to `stdout`, the `printNl` appends a newline to the end of the printed result.

3.2.3 The system message

`SystemDictionary system: aString`
: This message invokes system(3), passing the string `aString` to it. It returns an integer which is whatever system(3) returns.

3.2.4 The getenv message

`SystemDictionary getenv: aString`
: This message does a getenv(3) on string `aString`, and returns the result. The result is either a string, or `nil` if there was no environment variable with the name `aString`.

3.2.5 Initialization blocks

`SystemDictionary addInit: aBlock`
: This message adds *aBlock* to the set of init blocks that the system has. Init blocks are invoked after loading binary image files or kernel files, but before loading any command line files or reading from standard input. *aBlock* should be a block that takes no arguments.

3.2.6 Editing method definitions

You can have GNU Smalltalk invoke GNU Emacs to edit the source code definition of a method. You must have an explicit load of the `st.el` file in your `.emacs` file in order to do this. See Section 4.1 [Autoloading], page 21.

If you send the method's class the `edit:` message, and pass the method's name as the argument, and if you can invoke GNU Emacs by typing `emacs`, you should be able to exercise this feature.

Example

```
"Edit the definition of the edit: method itself!"
Behavior edit: #edit: !
```

Note that this only allows you to visit the method definition; this does not cause the method to be redefined after you edit it. However, it is conceivable that could create a method that does this in a fairly simple way.

3.2.7 Explicitly loading files

The `fileIn:` message sent to the FileStream class, with a file name as a string argument, will cause that file to be loaded into Smalltalk.

For example,

```
FileStream fileIn: 'foo.st' !
```

will cause `foo.st` to be loaded into GNU Smalltalk.

3.2.8 Memory accessing methods

GNU Smalltalk provides methods for directly accessing real memory. You may access memory either as individual bytes, or as 32 bit words. You may read the contents of memory, or write to it. Due to the limitations of Integers in GNU Smalltalk, you can only deal with word memory as 31 bit quantities.

You may also determine the real memory address of an object or the real memory address of the OOP table that points to a given object, by using messages to the `Memory` class, described below.

There are two basic classes which provide methods to access memory: `ByteMemory` and `WordMemory`. The methods for these classes are as follows:

`ByteMemory class at: address`
: Returns the byte at *address* as an `Integer`. *address* is also an `Integer`.

`ByteMemory class at: address put: value`
: Sets the byte at *address* (an `Integer`) to be *value* (an `Integer` in the range of 0 to 255).

`WordMemory class at: address`
: Returns an `Integer` that represents the contents of *address*, which should also be an `Integer`. Depending on the machine architecture, you'll probably want to ensure that *address* is a multiple of 4. Note that this method does not currently work on little endian architectures (such as a VAX).

`WordMemory class at: address put: value`
: Stores the `Integer` *value* into real memory at *address*. Same alignment cautions as above, and also doesn't currently work on little endian architectures.

`Bigendian`
: A global variable that is set to `true` on machine architectures where the most significant byte of a 32 bit integer has the lowest address (e.g. 68000 and Sparc), and false on architectures where the least significant byte occurs at the lowest address (e.g. VAX).

To find out the real memory address of an object or its OOP table entry, you may use one of the methods described below.

`Memory class addressOfOOP: anObject`
: Returns the address of the OOP for *anObject*. The address is an *Integer* and will not change over time (i.e. is immune from garbage collector action).

`Memory class addressOf: anObject`
: Returns the address of the actual object that *anObject* references. Note that this address is only valid until the next GC flip; thus it's pretty risky to count on the address returned by this method for very long.

3.2.9 Producing backtraces

`SystemDictionary backtrace`
: When you send this message to a system dictionary (i.e. `Smalltalk`, the only system dictionary available), it produces a backtrace of the methods that are the ancestors of the currently executing method. This backtrace is exactly like the one that is printed when the interpreter encounters an error.

3.2.10 Controlling tracing of bytecode execution

`SystemDictionary executionTrace: aBoolean`
: This method controls whether byte codes are printed as the interpreter processes them or not. It has the exact same effect as the command line switch `-e`, but allows for very fine grain control of the tracing. Use `true` to enable tracing, and `false` to disable tracing.

3.2.11 Printing Smalltalk stack during execution

`SystemDictionary verboseTrace: aBoolean`
: This method is used in conjunction with `SystemDictionary executionTrace:`. When given the argument `true`, and execution tracing is on, it causes the top element of the Smalltalk stack to be printed before each byte code is executed.

3.2.12 Assistance using C debuggers

`SystemDictionary debug`
: This primitive method calls a C routine which is called `debug`. This can be useful in the following way: you want to get access to dbx or gdb at a particular point in the execution of a Smalltalk method. There typically is no easy way to do this. In the GNU Smalltalk system, the C routine called `debug` exists solely so that you can put a breakpoint in it from a debugger such as dbx (it is an empty routine, and thus harmless). You put a call to the C `debug` routine in the C function that you want to debug.

  So, if you put a breakpoint in the `debug` routine, and invoke the `debug` method just before the code which you want to debug, you gain control for detailed debugging.

3.2.13 Control C profiling during execution of methods.

`SystemDictionary monitor: aBoolean`

This primitive method allows the user to enable or disable the generation of C profiling information during the operation of the interpreter. You must modify the `Makefile` to switch the compilation to profiling mode and recompile the entire system for this to take effect.

When you pass `true` to this method, you enable the collection of profiling data during the operation of the interpreter to be later analyzed by the `gprof` program. Passing `false` disables this collection. You can use this facility to closely monitor the operation of the interpreter over a given set of Smalltalk code to see where it's spending its time.

3.2.14 Controlling generation of GC flip messages

`SystemDictionary gcMessage: aBoolean`

This message allows the user to control whether a message is printed each time the garbage collector flips between old and new space. The default state is `true`, meaning that when the garbage collector performs a flip, a message will be printed. If you supply `false` as the argument of this message, the message generation is disabled until you turn it back on.

3.2.15 Explicit termination of GNU Smalltalk

`SystemDictionary quitPrimitive`

Invoking this method causes an immediate and unconditional exit from GNU Smalltalk.

3.2.16 Alternate assignment operator

`:=`

GNU Smalltalk allows the use of `:=` as an alternative assignment operator in addition to the standard `_`. This is for compatibility with other Smalltalk implementations. Be sure to surround this operator with at least one space on each side; failure to do so will cause parse errors.

3.3 Differences from Blue Book Smalltalk

The following is a brief description of the differences and omissions between GNU Smalltalk and "Blue Book" Smalltalk (Smalltalk-80).

- No long integers (yet)
- No fractions (yet)
- Time values based on UNIX epoch of Jan 1, 1970 (although Date instances *are* based on the Smalltalk base date of January 1, 1901).
- Millisecond times are since midnight, instead of since the "millisecond clock" turned over.
- No async signals (yet)
- Beginnings of the technology for a graphical user interface, and beginnings of an incremental development environment, but nowhere near what commercial Smalltalks provide.

- Class Delay is not implemented (yet)

4 Smalltalk interface for GNU Emacs

GNU Smalltalk comes with its own Emacs mode for hacking Smalltalk code. It also provides tools for interacting with a running Smalltalk system in an Emacs subwindow.

4.1 Autoloading GNU Smalltalk mode

To cause Emacs to automatically go into Smalltalk mode when you edit a Smalltalk file (one with the extension `.st`), you need to add the following lines to your .emacs file:

```
(setq auto-mode-alist
      (append  '(("\\.st$" . smalltalk-mode))
       auto-mode-alist))

(autoload 'smalltalk-mode "~/smalltalk-1.1.1/st.el" "" t)
```

This presumes that you have placed Smalltalk as a subdirectory of your home directory; if you have placed it somewhere else, you'll need to change the file name mentioned in the `autoload` line accordingly.

If you want additional speed, you can byte compile the `st.el` file, and change the autoload line to refer to `st.elc` instead.

If you plan on using the `edit:` method to edit the source code of your methods, you'll want to explicitly load in the `st.el` or `st.elc` file instead of letting them autoload in. Use

```
(load "~/smalltalk-1.1.1/st.el")
```

instead of the autoload line.

It is also a good idea to put the main GNU Smalltalk directory in your EMACSLOADPATH environment variable.

4.2 Smalltalk editing mode

The GNU Smalltalk editing mode is there to assist you in editing your Smalltalk code. It tries to be smart about indentation (if you use newline at the end of a line). Also, if you want to re-indent a line, use *M-Tab*. Since Smalltalk syntax is highly context sensitive, the Smalltalk editing mode will occasionally get confused when you are editing expressions instead of method definitions. In particular, using local variables, thus:

```
| foo |
    foo _ 3.
    ^foo squared !
```

will confuse the Smalltalk editing mode, as this might also be a definition the binary operator `|`, with second argument called *foo*. If you find yourself losing when editing this type of expression, put a dummy method name before the start of the expression, and take it out when you're done editing, thus:

```
x
| foo |
    foo _ 3.
    ^foo squared !
```

4.3 Smalltalk interactor mode

Several new features have been added to the Smalltalk editing mode for GNU Emacs. The most exciting one is the Smalltalk interactor system, which basically allows you run in GNU Emacs with Smalltalk files in one window, and Smalltalk in the other. You can, with a single command, edit and change method definitions in the live Smalltalk system, evaluate expressions, make image snapshots of the system so you can pick up where you left off, file in an entire Smalltalk file, etc. It makes a tremendous difference in the productivity and enjoyment that you'll have when using GNU Smalltalk.

To start up the Smalltalk interactor, you must have a working GNU Smalltalk interpreter somewhere in your `PATH` environment variable, and be running in GNU Emacs. You should be in a buffer that's in Smalltalk mode (which can be automatically enabled by adding the proper magic to the auto-mode-alist, see Section 4.1 [Autoloading], page 21). For this example, let's use the file `t.st`. Visit this file, make sure that you're in Smalltalk mode, and type *`C-c m`*. A second window will appear with GNU Smalltalk running in it.

This window is in most respects like a Shell mode window. You can type Smalltalk expressions to it directly and re-execute previous things in the window by moving the cursor back to the line that contains the expression that you wish to re-execute and typing return.

Notice the status in the mode line (e.g. '`starting-up`', '`idle`', etc). This status will change when you issue various commands from Smalltalk mode.

When you first fire up the Smalltalk interactor, it puts you in the window in which Smalltalk is running. You'll want to switch back to the window with your file in it to explore the rest of the interactor mode, so do it now.

Let's try executing a range of code first. Mark the region around:

```
('Welcome to GNU Smalltalk [', Version, ']

This file contains a wealth of goodies, not all packaged neatly.
It sort of grows by accretion, so you''re likely to find most
anything in here.' ) printNl.
```

Now type *`C-c e`*. The expression in the region is sent to Smalltalk and evaluated. The status will change briefly to indicate that the expression is executing. This will work for any region that you create. If the region does not end with an exclamation point (which is syntactically required by Smalltalk), one will be added for you.

As a second example, move the cursor down to the region of code that looks like:

```
Object withAllSubclasses do:
    [ :subclass | (subclass name notNil and: [ subclass comment isNil ])
                      ifTrue: [ subclass name print.
                                ' has no comment.' printNl ]
                      ]

!
```

This code will find any class that doesn't have a comment associated with it (I used this to track down classes that needed commenting). Put the cursor somewhere between the

first and last lines and type *C-c d*. The entire expression will be sent to GNU Smalltalk, and after scanning all the classes, GNU Smalltalk will report that class Delay has no comment. This command (also invokeable as *M-x smalltalk-doit*) uses a simple heuristic to figure out the start and end of the expression: it searches forward for a line that begins with an exclamation point, and backward for a line that does not begin with space, tab, or the comment character, and sends all the text in between to Smalltalk. If you provide a prefix argument (by typing *C-u C-c d* for instance), it will bypass the heuristic and use the region instead (just like *C-c e* does).

Now move a ways down to some text that looks like

```
!BlockContext methodsFor: 'debugging'!

callers
    self inspect.
    caller notNil
        ifTrue: [ caller callers ]
!
```

Put the cursor on the line containing `self inspect.`. Type *C-c c*. Nothing much will appear in the Smalltalk window, but what you've done is you've compiled the BlockContext>>callers method. To test this, you can switch to the Smalltalk window and type

```
[ 'foo on you' ] callers!
```

which will produce a simple backtrace of the invocation stack.

The *C-c c* command uses a similar heuristic to determine the bounds of the method definition. Typically, you'll change a method definition, type *C-c c* and move on to whatever's next. If you want to compile a whole bunch of method definitions, you'll have to mark the entire set of method definitions (from the `methodsFor:` line to the `!!`) as the region and use *C-c e*.

After you've compiled and executed some expressions, you may want to take a snapshot of your work so that you don't have to re-do things next time you fire up Smalltalk. To do this, you use the *C-c s* command, which creates a Smalltalk binary image called `mst.im`. If you invoke this command with a prefix argument, you can specify a different name for the image file, and you can have that image file loaded instead of the default one by using the `-I` flag on the command line when invoking Smalltalk.

You can also evaluate an expression and have the result of the evaluation printed by using the *C-c p* command. Mark the region and use the command.

To file in an entire file (perhaps the one that you currently have in the buffer that you are working on), type *C-c f*. You can type the name of a file to load at the prompt, or just type return and the file associated with the current buffer will be loaded into Smalltalk.

When you're ready to quit using GNU Smalltalk (assuming that that happens sometimes), you can quit cleanly by using the *C-c q* command. If you want to fire up Smalltalk again, or if (heaven forbid) Smalltalk dies on you, you can use the *C-c m* command, and Smalltalk will be reincarnated. Even if it's running, but the Smalltalk window is not visible, *C-c m* will cause it to be displayed right away.

You might notice that as you use this mode, the Smalltalk window will scroll to keep the bottom of the buffer in focus, even when the Smalltalk window is not the current window.

This was a design choice that I made to see how it would work. On the whole, I guess I'm pretty happy with it, but I am interested in hearing your opinions on the subject.

Speaking of opinions, the whole Smalltalk interactor mode is quite young (about a week old as I write this) and still could use some maturation. If you have comments or ideas about how to improve this system, please let me know. I have been planning something like this for a while, but even with a vision in my head of how it was going to work, I was unprepared for just how much of a difference it makes when hacking Smalltalk. Hacking on STIX was infinitely easier once I got this system operational.

One minor thing that you may note when using the interactor: if you try to get the source code for a method that's been compiled from the interactor, you will get garbage. This is because I create temporary files to hold the method definitions and send a `fileIn:` message to cause them to be loaded. The method definitions will point to this temporary file as their source code instead of the real source code. This is not too hard to fix, and I felt that it was an acceptable tradeoff to make in order to get the 1.1 release out as soon as possible.

5 Smalltalk Interface to X (STIX)

This section describes STIX: the Smalltalk Interface to X.

5.1 What is STIX?

This version of GNU Smalltalk comes with a simple interface to X window. This interface is currently pretty much a direct interface to the X protocol layer...it's even lower than Xlib. What it does provide, however, is a more object oriented framework for dealing with Xlib objects.

STIX is an initial cut at an interface from Smalltalk to X. It is intended to be usable for simple things, and my hope is that someone will help me out by filling in the missing parts of the X protocol interface. Even with a later version of Smalltalk that contains call-ins from C, the need for an interface at this level will persist.

STIX is somewhat of an experimental interface: the mapping of X Window protocol functions to Smalltalk is not entirely straightforward. Thus, you may find that the current implementation of STIX is not the cleanest way to provide such an interface. It is hoped, however, that the next release will be substantially better organized, and the class hierarchy will be more well defined.

STIX also has a simple implementation of the `Pen` class that is described in the Blue Book...it is used to draw some of the graphics in the example.

5.2 Requirements for STIX

To be able to run STIX, you need to have a working version of X11 release 4 installed and running on your machine. You must know where the include files for X can be found (often, this is `/usr/include/X11`).

5.3 Running STIX

The STIX example has been designed to be as easy as possible to run. You should perform the following steps:

1. Change your working directory to be the `stix` subdirectory of the Smalltalk directory.
2. Configure GNU Smalltalk using `config.mst`, as you did before compiling GNU Smalltalk in the parent directory.
3. Edit the `Makefile` file and change the definition of `XINCLUDE` to point to the directory that contains the include files for X.
4. Compile GNU Smalltalk by typing '`make`'.
5. Load and run the STIX example by typing

   ```
   mst -Vi t.st
   ```

   This will load normally, except that CFuncs.st will take a bit longer than it normally does to load. The first time that you do this, you should see each of the kernel files being loaded; if you don't, it means that you are using a saved image, which won't have the STIX methods loaded in it. Once it's loaded and the binary image is saved, you should see several messages indicating that things are being executed, and, if you're successful, a window will appear that will let you know that it's from Smalltalk. *Be*

sure to use the `-i` *flag only the first time you build the STIX image; once it's built, you don't want to use it again as it tells Smalltalk to ignore any image file it finds and rebuilt the image from scratch.*

6 Future directions and tasks for GNU Smalltalk

Presented below is the set of tasks that I feel need to be performed to make GNU Smalltalk a more fully functional, viable system. They are presented in no particular order. I would *very much* welcome any volunteers who would like to help with the implementation of one or more of these tasks. Please write to me, Steve Byrne, currently at `sbb@eng.sun.com` if you are interested in adding your efforts to the GNU Smalltalk Project.

Tasks:

- Port to other computers/operating systems. The code thus far has shown itself to be relatively portable to various machines and UNIX derivatives. The architecture must support 32 bit pointers and long integers, and have allocated addresses that are either all positive or all negative. You'll probably want a reasonable amount of paging area, as GNU Smalltalk currently uses a fair amount of memory while it's running.
- Comment the C code more thoroughly. The C source code for GNU Smalltalk could stand a more thorough commenting. This includes things like describing the operation of the byte code interpreter.
- Comment the Smalltalk code more thoroughly. In order to be more useful to neophyte Smalltalk users, the method definition code should be commented to at least describe the behavior of each method; ideally, the method definitions would have normal internal commentary as well.
- Create a portable version of the Delay class primitive. This depends on the implementation of asynchronous signals, and needs to use some kernel call to set up an interrupt after the appropriate number of milliseconds. Unfortunately, I do not have access to a true System V UNIX box to find out what kernel call allows me to do this (BSD has setitimer(2)).
- Add an instance variable `Block-` and `MethodContext`s that contains the actual class in which the method was found. This will be used to improve the backtrace printing code so that users can see how each message was resolved by seeing the class that the executing method was finally found in. Block contexts don't really need this, but they should have the same number and organization of instance variables. Perhaps, the vague references in the Blue Book about a `ContextPart` class refer to a class that's the parent of both `BlockContext` and `MethodContext` that holds the common instance variables; this would probably be a variable in that class.
- Write a debugger for Smalltalk. You may note vestiges of this in this release. Basically, my thinking is that the debugger would be written in Smalltalk, and would allow for setting breakpoints at byte codes, getting method invocation backtraces, moving up and down the invocation stack, inspecting and altering variables, possibly allowing for source level debugging (needs cooperation from compiler), etc. I am willing to hear other ideas on the subject; my mindset is inspired by the debugging capabilities of present day Common Lisp systems. Perhaps this debugger could be integrated with the Emacs editing mode to provide a simple way to indicate where to put breakpoints and inspect variables.
- Add more test cases to the test suite. The test suite is much too small. Ideally, after making changes to the Smalltalk system, you'd put it through its paces by running with the test suite and make sure that you hadn't broken anything.

- Write a C call-in to Smalltalk. The idea here is that C functions should be able to send messages in Smalltalk, and receive results, which allows for true bi-directional Smalltalk<=>C communication. Should allow for C signals to signal Smalltalk semaphores, etc.
- Make Smalltalk more interfaceable to C: allow a C program to be the main program, and it merely has to call initSmalltalk() before it tries to do Smalltalk things.
- Make Smalltalk C++ compatible (able to compile with C++). In order to allow people to use C++ code with GNU Smalltalk, it would be convenient to have the GNU Smalltalk system be entirely compilable with G++.
- Switch to using getopt. GNU Smalltalk currently uses a home-grown option parser; this is primarily a standardization task.
- Write a byte-code Smalltalk compiler in Smalltalk. As a first step towards creating a full Smalltalk compiler, duplicate the functionality provided by the internal byte code compiler.
- Convert byte-code compiler to compile to machine code, possibly using the GCC back end (if feasible). My thinking is to capitalize on some of the compiler technology that's used in Self to get high quality generated code.
- Improve the byte-code interpreter's efficiency. Until the real compiler is completed, for machine architectures that the code generator does not yet know about, and to make for simple porting and bootstrapping, we will need a viable byte-code interpreter. The current interpreter could stand a bit of tightening up.
- Implement LargeIntegers for Smalltalk.
- Implement Fractions for Smalltalk.
- Add machine/os type as a variable in Smalltalk dictionary. Perhaps via a `#define` in the the configuration files, this would manifest itself as a method or global variable in the Smalltalk dictionary. It might return a `String`, such as `"Sun3os4"`, or a `Symbol`, or perhaps have a couple of different values: machine type, operating system version, etc.
- Design a conditional compilation mechanism, maybe like #+VAX in Common Lisp. I am not entirely sure how this would work given Smalltalk's syntax. We can do a poor-man's implementation today by conditionally doing `fileIn`s of specific pieces of code, but this is a little more crufty than is desirable.
- Switch from OOP table representation to direct pointer representation. Currently, all references to objects point to an entry in an indirect table, called the OOP table. This has advantages in that objects may be moved around (i.e. during garbage collection) without having to worry about who has pointers to the object. It also allows for simple enumeration of all instances of a particular class, and allows for a simple implementation of the `become:` message. However, most present day Smalltalk implementations have gotten away from this representation, and instead directly reference objects.
- Investigate switching to a generational garbage collector. Many of Smalltalk's kernel objects are relatively static, and having to sweep past them when garbage collecting takes time. Also, some objects are very fleeting indeed (such as `BlockContexts` and `MethodContexts`), and shouldn't unnecessarily slow down garbage collection. Also worth investigating is not creating real objects for methods; rather, have a method

cache (stack) that contains "proto-methods". Check at certain places for storing a pointer to a method that's a proto-method and migrate it to be a real one when that happens.

- Improve the interactive inspector in GNU Emacs. Things to add include inspect this object/expression; examine the selectors for this calls; fixup the handling of the temp files so that file-outs of methods that have been incrementally redefined don't lose big-time. Also, editing object contents on the screen might be nice. This would make the generally available GNU Smalltalk mode more and more like traditional Smalltalk development environments.
- Flesh out the STIX system. There are still a number of the protocol items that have yet to be coded; once they're done, work can begin on a higher level interface (Xlib or above). Things above Xlib might include menus and text type windows (things such as you might find in Xt).
- Create an X Window development system for Smalltalk. This would be a window-based browser, similar to that which is available in commercial implementations of Smalltalk.
- Extend the `quitPrimitive` method to also take an argument, which is the value to pass to the `exit()` system call.
- Add other interesting example classes, such as Directory, or Socket, i.e.

```
"Establish a connection to the mail system on PREP"
sock _ Socket onHost: 'prep.ai.mit.edu' port: 25.
'HELO RAMBO' printOn: sock.
sock nl.
...

(could also have symbolic names for the port numbers, but you get
the general idea...)
```

7 Acknowledgements

I would like to publically thank the following people, who have helped out in one way or another since with this release (and the 1.2 release) of GNU Smalltalk:

Fritz Nordby, Michael Mellinger, Doug McCallum, Karl Berry, Dave Bodenstab, Mark Wadsworth, Bill Trost, William Cook, Trip Becket, Alan Knight, Alistair Grant, Michael Richardson, Andrew Gelsey, Kevin Hester, David MacKenzie, Doug Peters, Michael Bushnell, Florin Spanachi, David England, Charles Johnson, David B. Serafini, R James Noble, David Duke, Mark S. Johnson, Gary Campbell, Peter Dobcsany, Kent Williams, Wilson Ho, Karl Kleinpaste, Len Tower, Paul Regenhardt, Joe Pallas, Peter Kropf, Mark Bush, Per Bothner, Kevin Rigotti, Lance Norskog, Pascal Meheut, Richard Goerwitz, Horst Duchene, Olivier Blanc and especially Jeff Baird.

(I apologize if I've left anyone out).

Some have contributed code, others have helped out with discussing implementation issues, and others have done ports to various machines. *I appreciate all of your efforts!!!*

I must also thank the FSF for providing a wonderful set of tools with which GNU Smalltalk was built. Although I don't do it currently, I believe it is the case that all of the tools that are needed for the construction and development of GNU Smalltalk are now available from FSF (including make, tar, diff, and other random tools). My world is a significantly better place thanks to the hard work of GNU people. *THANK YOU!!!!*

Table of Contents

Introduction **1**

1 Installation **3**
1.1 Which files to examine before compiling 3
1.2 Compiling GNU Smalltalk 3
1.3 Implementations 4
1.4 Readline interface for GNU Smalltalk 5

2 Invoking GNU Smalltalk **7**
2.1 Command line arguments 7
2.2 Startup sequence 8
2.3 Syntax of GNU Smalltalk 9
2.4 Operating GNU Smalltalk 10
2.5 Per-user init files 10
2.6 Running the test suite 10

3 Features of GNU Smalltalk **11**
3.1 General Features 11
3.2 Additional features of GNU Smalltalk 11
3.2.1 Using the C callout mechanism 12
3.2.2 UNIX file-IO primitive messages 14
3.2.3 The system message 15
3.2.4 The getenv message 15
3.2.5 Initialization blocks 15
3.2.6 Editing method definitions 15
3.2.7 Explicitly loading files 16
3.2.8 Memory accessing methods 16
3.2.9 Producing backtraces 17
3.2.10 Controlling tracing of bytecode execution 17
3.2.11 Printing Smalltalk stack during execution 17
3.2.12 Assistance using C debuggers 17
3.2.13 Control C profiling during execution of methods. 18
3.2.14 Controlling generation of GC flip messages 18
3.2.15 Explicit termination of GNU Smalltalk 18
3.2.16 Alternate assignment operator 18
3.3 Differences from Blue Book Smalltalk 18

4 Smalltalk interface for GNU Emacs **21**
4.1 Autoloading GNU Smalltalk mode 21
4.2 Smalltalk editing mode 21
4.3 Smalltalk interactor mode 22

5 Smalltalk Interface to X (STIX) 25
5.1 What is STIX? 25
5.2 Requirements for STIX 25
5.3 Running STIX 25

6 Future directions and tasks for GNU Smalltalk .. 27

7 Acknowledgements 31