A QUICK TOUR OF SABER-C

Version 1.2

1.1. Introduction

Saber-C is an interactive programming environment for the C language. It helps programmers develop, test, and debug C programs. With Saber-C, users can interpret and debug their programs as they are being developed, without having to use a compiler or a debugger.

This manual provides new users with a quick introduction to the features and commands of Saber-C. The reader should have some familiarity with the C programming language and the UNIX operating system.

1.2. Starting and Exiting Saber-C

To start Saber-C, enter the command **saber** to the shell.

```
% saber

        Saber-C (Version 1.2)
  Copyright (C) 1986, 1987 by Saber Software

Attaching: /lib/libc.a

Do you want instructions? n

1 ->
```

When **saber** starts up, it examines the current working directory and the user's home directory for the file .saberinit. If the file is not found, **saber** prompts for instructions. If you intend to continue reading this manual, you should skip the on-line instructions. (The instructions prompt can be eliminated by creating a file called .saberinit in your home directory. The file can be empty.)

To exit **saber**, use the quit command.

```
1 -> quit();

%
```

1.3. Getting Help

The help command provides on-line information on how to use all the **saber** commands. To see a list of all the commands, call help. To get information about a particular command, call help with the name of the command.

```
1 -> help();
/* displays a list of all commands */

2 -> help("load");
/* displays information on the load command */
```

1.4. Interacting with Saber-C

When **saber** is ready to accept input from the user, it displays a line number followed by a -> prompt. All input to **saber** must consist of proper C expressions or statements. **Every expression or statement must be followed by a semi-colon.**

```
1 -> int i;
2 -> i = 100;
(int) 100
3 ->
```

Saber uses different prompts to indicate different input conditions. If the user enters an incomplete expression or statement — usually because of a missing semi-colon or an incomplete block — the +> prompt is used.

```
1 -> 123 +
2 +> 456 +
3 +> 789;
(int) 1368
4 ->
```

1.5. Executing C in the Environment

Saber's environment will accept and execute the entire C language. The user can declare variables, create types, and use the C preprocessor.

```
1 -> #define ARRAY_SIZE 10
2 -> struct pair { int x ; double d ; } ;
3 ->
4 -> struct pair list[ARRAY_SIZE];
5 ->
```

Functions can be declared directly in the environment. When defining a function, the return type of the function must be specified. For example, the return type `int` for the function `factorial` must be specified when entering the function, even though both **saber** and compiler do not require this of functions defined in a file.

```
1 -> int factorial(n)
2 +> int n ;
3 +> {
4 +>     if( n > 1 ) return (n * factorial(n-1)) ;
5 +>     else return 1 ;
6 +> }
7 ->
```

Although the user can write entire programs in the environment, it is difficult to modify and debug source code that has been entered directly in the environment. The user should place program code in a file so that it can be easily edited and loaded.

1.6. Executing Commands

All **saber** commands are implemented as C functions that can be executed directly from the environment. The user executes a command by calling it as a function. Any options or additional information for the command are passed as arguments to the function.

```
1 -> /* Call load with a file as the argument */
2 -> load("main.c") ;
Loading: main.c
(int) 0
3 ->
```

Since commands are executed as functions, they must always be followed by an open parentheses, a close parentheses, and a semi-colon, even if the command is being called without an argument.

```
4 -> /* Calling load without an argument */
5 -> load();
load: All loaded modules are up to date.
(int) 0
6 ->
```

1.7. Responding to Errors and Warnings

Saber checks all input, either from the terminal or from a file, for syntax errors and static violations. When an expression, statement, or program is executed, **saber** checks it for possible run-time violations.

Benign violations are reported as warnings, from which the user can continue. More serious violations are reported as errors; the user usually cannot continue from errors.

When a violation is reported, the user is presented with a list of options. For example, at the error shown below, the user can break execution at the location of the violation, quit execution of the statement, or edit the line containing the violation.

```
1 -> extern int test() ;
2 -> test() ;
Error#61: Calling undefined function test().
Options: break/quit/edit [b] ?
```

The default choice, which is selected if the user enters a carriage return, is enclosed within the square brackets.

1.8. Creating Break Levels

A break level is is generated when the user breaks execution at an error or warning. (Break levels can also be generated with the `stop` command, which is discussed later.) Break levels allow the user to examine or correct a violation before continuing execution.

In the example below, the user chose to break execution at the location where the undefined function is called.

```
2 -> test() ;
Error#61: Calling undefined function test().
Options: break/quit/edit [b] ? b

To continue, use 'cont(int)'. If you define the
function, use 'cont()' or <Control-D> to retry.
(break 1) 3 ->
```

At a break level, the user can perform nearly any action that is permitted at the top level of the environment, as well as examine the execution stack by using the where command.

```
(break 1) 3 -> where() ;

break #1, line 3, caused by error #61 (Undefined function)
top level, line 2.
(void)
(break 1) 4 ->
```

1.9. Leaving a Break Level

The message that is displayed when the break level is generated indicates how the user can continue execution.

One way to continue execution from the break level generated in the previous example would be to call the cont command with an integer value. This value would represent the return value of the undefined function test.

```
(break 1) 3 -> cont(99);
(int) 99
4 ->
```

Another way to continue execution would be to define the function test at the break level and call cont without an argument. A shortcut for calling cont without an argument is to enter a <Control-D>.

```
(break 1) 3 -> int test() { return 1234;}
(break 1) 4 -> ^D
(int) 1234
5 ->
```

The reset command can be used if the user wants to exit the break level without continuing execution.

```
(break 1) 3 -> reset() ;
Resetting to top level.
4 ->
```

1.10. Loading Files

Users can load source code files, object code files (with and without debugging information), and library files into **saber**. A file can be loaded two ways: it can be specified on the command line when **saber** is invoked or it can be loaded interactively from within **saber**.

The example below displays the listing for the file calc.c which contains the functions main and calc. This program takes one argument that tells it how many values to calculate. For every calculation, it prompts the user for a number and prints the square root and log of the number.

```
#include <math.h>

void calc() ;
float num ;

main(argc, argv) int argc ; char **argv ;
{
    char buf[100] ;

    printf("Please enter a number: ") ;
    gets(buf) ;
    num = atof(buf) ;
    calc(num) ;

    exit(0) ;
}

void calc(val) float val ;
{
    printf( "val = %f, square root = %f, log = %f\n",
        val, sqrt(val), log(val) ) ;
}
```

The example below shows how to load the source code file and the system math library from the command line. (The -w switch is used to suppress reporting of any warnings when loading the files.)

```
% saber -w calc.c -lm


       Saber-C (Version 1.2)
  Copyright (C) 1986, 1987 by Saber Software

Attaching: /lib/libc.a
Loading: -w calc.c
Attaching: -w /usr/lib/libm.a
```

These files could be loaded interactively, after entering **saber**, by using the load command.

```
1 -> load(" -w calc.c -lm") ;

Loading: -w calc.c
Attaching: -w /usr/lib/libm.a
```

Any options or switches that specify how the files should be loaded are included within the argument to the load command. In the example above, the user specified that the files should be loaded with the -w option, which suppresses the reporting of warnings (similar to the compiler).

If the load command is called with the name of a library, it *attaches* the library, thereby making the contents of the library available for use. The modules of the library are linked dynamically when they are needed. For example, the standard library function printf is linked from the library when it is used.

```
2 -> printf("Done loading.\n") ;
Linking from /lib/libc.a ...Linking completed.
Done loading.
(int) 14
3 ->
```

Once a function has been linked from a library, it does not need to be linked again.

1.11. Executing Programs

Saber supports two modes of execution: direct execution of functions from the environment and execution of programs with processing of command line arguments.

Functions may be executed as soon as they are defined, even if they contain references to undefined variables. When a function is executed, all loaded libraries are searched in order to satisfy references to undefined variables and functions.

```
3 -> calc(100.0) ;
Linking from /lib/libm.a -w ...Linking completed.
Linking from /lib/libc.a ...Linking completed.
val = 100.000000,square root = 10.000000,log = 4.605170
(void)
4 ->
```

The run command is used to mimic the behavior of a program as if it were being executed from the shell. Run reinitializes all global variables, executes main with "command line" arguments, and closes all input/output buffers and file descriptors after main exits. If **saber** is executing within a windowing system, run will also open a separate window for handling program input and output.

```
4 -> run() ;
Executing: a.out
Linking from /lib/libc.a ...Linking completed.
Please enter a number: 10
val = 10.000000,square root = 3.162278,log = 2.302585
Program exiting with return status = 0.
Resetting to top level.
5 ->
```

Run constructs its arguments for main from the "command line" passed to it. If this argument is missing, command line arguments from the previous call to run are re-used. To execute main without command line arguments, use an empty string as the argument to run.

1.12. Using the Debugging Commands

Saber lets users set breakpoints, which interrupt execution at a specific location in a program, and watchpoints, which interrupt execution when an address is modified.

A breakpoint is set by calling the stop command with a line number, a filename and a line number, or a function.

```
5 -> stop(calc);
Breakpoint #1 set at "calc.c" , line 17, calc().
6 ->
```

When the breakpoint is reached, execution stops, and a break level is created. All source lines at which breakpoints are set are preceded by the character B. The source line at which execution was interrupted is preceded by a *.

```
6 -> run() ;
Executing a.out
Please enter a number: 144

Stopped in calc(val = (double) 1.440000e+02) at "calc.c", line 17
  16: {
*B 17: printf( "val = %f, square root = %f, log = %f\n",
  18:         val, sqrt(val), log(val) ) ;
(break 1) 7 ->
```

All active functions, variables, macros, and tag names may be examined directly from the break level. Execution can be continued by entering a <Control-D> or calling the cont command.

```
(break 1) 7 -> ^D
val = 144.000000,square root = 12.000000,log = 4.969813
Program exiting with return status = 0.
Resetting to top level.
```

The tag command can be used to interrupt execution when an address is modified. In the example below, a tag is set on the global variable num.

```
10 -> tag(&num);
Tag #1 set on 4 bytes beginning at address 0x89448.
(int 1)
11 ->
```

When the tagged address is modified, execution will be interrupted.

```
11 -> run() ;
Executing: a.out
Please enter a number: 99

Stopped in main(argc = (int) 1, argv = (char **) 0x89ba0)
at "calc.c" , line 14
  13: gets(buf) ;
* 14: num = atof(buf) ;
      ***tag #1 triggered.***
  15: calc(num) ;
(break 1) 11 ->
```

All debugging actions can be listed with the status command and deleted with the delete command. Delete should be called with the number of the debugging action.

```
(break 1) 11 -> status();
 (1) Tag: addresses 0x89448 - 0x8944b (location 'num').

(void)
12 -> delete(1) ;
Tag #1 deleted.
13 ->
```

The commands action and tagaction are versions of stop and tag that do not interrupt execution when triggered. Instead, they will execute an arbitrary action specified by the user. For example, action can be used to create conditional breakpoints by combining a C statement with the stop command

```
13 -> action(calc);
Setting action at "calc.c" , line 19, calc().
Use braces for multiple statements. <Control-D> to quit,
   <Control-C> to redo.
action -> if( num == 0.0 ) stop() ;
Action #1 set.
15 ->
```

Execution will be interrupted only if the value of num equals 0. In the example below, num equals 123 for the first execution of the program. When the program is executed again, num equals 0, and execution is interrupted by the action.

```
16 -> run() ;
Execution a.out
Please enter a number: 123
val = 123.000000,square root = 11.090537,log = 4.812184
Program exiting with return status = 0.
Resetting to top level.
17 ->
18 -> run() ;
Please enter a number: 0
Stopped in action #1, line 1, set in
calc(val = (double) 0.000000e+00) at "calc.c", line 19
   18: {
*<A1>   1: if( num == 0.0 ) stop() ;
   19:  printf( "val = %f, square root = %f, log = %f\n",
   20:          val, sqrt(val), log(val) ) ;
(break 1) 19 ->
```

1.13. Editing and Reloading

Files can be edited from within **saber** by using the edit command. Edit can be called with the name of a file, the name of a function, or a line number of a file. The example below will start the editor on the first line of the function calc.

```
20 -> edit(calc)  ;
```

Edit uses the editor specified by the shell variable **EDITOR**; if this variable is unset, the editor **vi** is used.

If **saber** is being run in a windowing environment, the edit job will be placed in a separate window. If the user is working on an ascii terminal, **saber** will be suspended while the editing job is active.

Once the file being edited has been modified and saved, the user can enter a <Control-Z> (suspend signal) to notify **saber** that the file is ready to be reloaded. If **saber** is being used with a windowing system, the file will be automatically reloaded; otherwise, the user will be prompted to reload the file.

If an error or warning is detected while the file is being reloaded, the editor will be updated to the location of the violation. This mechanism allows the user to correct all syntax errors and static violations without leaving the editor.

Another mechanism to reload modified files is with the load command. If load is called without an argument it will reload any modules that have been modified since they were loaded into **saber**. If load is called with the name of a file that has already been loaded, the file will be reloaded if it has been modified.

```
21 -> load()  ;
load: All loaded  modules are up to date.
(int) 0
22 -> load("calc.c") ;
load: Module 'calc.c' is up to date.
(int) 0
23 ->
```

A source code module will be reloaded if the source file or any of the header files it includes have been modified. An object code module will be reloaded if the object file has been recompiled.

Table of Contents