GNU SETL User Guide

Edition 0.1.2, for GNU SETL Version 0.1.2
Updated 7 June 2009

by dB

Table of Contents

1 The `setl` command 1

1.1 Examples 1
1.2 The `setl` command and arguments 2
1.3 #! invocation and '`-k`' 7
1.4 Environment variables 9
1.5 Signals 9
1.6 Exit status 9

2 The `setlcpp` command 10

2.1 Examples 10
2.2 The `setlcpp` command and arguments 11
2.3 Environment variables 12

3 The `setltran` command 13

3.1 Examples 13
3.2 The `setltran` command and arguments 14

Index 16

1 The setl command

The setl command is the primary interface to the GNU SETL system. In typical use, it preprocesses SETL programs using setlcpp, compiles them with setltran, and runs them on the GNU SETL virtual machine.

Depending on the specific run-time environment, your SETL program will be able to read from its standard input channel (stdin), write to its standard output and error channels (stdout and stderr), create and communicate with other processes, handle signals, listen on server ports, and open files and network connections.

1.1 Examples

This is typical output of the command 'setl --help' on a system with the GNU SETL Texinfo-based documentation properly installed:

```
GNU SETL programming language processor

Usage: setl [OPTIONS] [INPUT] [--] [ARGS]

  --[no]cpp           force [non]use of preprocessor
  -I..., -D..., -U...  passed to setlcpp; these imply --cpp
  --compile, -c       spew VM code on stdout, don't run
  --translated, -t    input is VM code, not SETL source
  --font-hints        just emit source prettyprinting hints
  --keyword-case=any|upper|lower    ("stropping" convention) -
                       control keyword recognition (default any)
  --identifier-case=any|upper|lower|mixed    control recognition
                       of user variable names (default any)
  --maxmem=N          limit memory use (k, m, or g suffix OK)
  --restricted, -r    restrict rights, for untrusted code
  --allow-open=FILE,MODE ...   restricted mode exemptions
  --allow-fd-open=FD,MODE ...   more restricted mode exemptions
  --help, -h          display this help on stdout and exit
  --version           display version info on stdout and exit
  --verbose, -v       make noise on stderr
  --debug             make more noise on stderr

  -FD                 input from numeric file descriptor FD
  |COMMAND            input from piped stdout of COMMAND
  FILENAME            input from file FILENAME
  STRING              get whole program directly from STRING
  -                   input from stdin (default if no input arg)

  --, -x              subsequent args are all for user pgm
  -k INPUT            for "#! ... -k" scripts

Examples:
```

```
  setl mypgm.setl -- my args
  setl 'print ("Hello, world.");'

See http://setl.org for more documentation, source code, etc.

The command "info setl" is also available on your system.

Please report bugs to bacon@cs.nyu.edu.  Thanks for using
SETL, the World's Most Wonderful Programming Language!
```

This is the output of the command '`setl "print(57);"`':

```
57
```

And of '`setl "print(command_line);" -- a "b c" 57`':

```
[a 'b c' '57']
```

1.2 The `setl` command and arguments

Here is the general form of the `setl` command:

```
setl [options] [input] [--] [run-time args]
```

The *options* include:

'`--[no]cpp`'

> By default, the GNU SETL Preprocessor, `setlcpp`, an adaptation of GNU CPP (the GNU C Preprocessor), is applied to any piece of program that appears to require it. Use this option to make an explicit choice. Options '`-I...`', '`-D...`', and '`-U...`' (see below), which are meaningful only to the preprocessor, also imply '`--cpp`'.
>
> The effective setting of this option in the absence of an explicit choice depends on whether the input segment appears to have possible `setlcpp` directives. Because a false positive is harmless, apart from incurring a little extra overhead for an unnecessary `setlcpp` invocation, the default is '`--cpp`' if there are any lines whose first token is '`#`'.
>
> There are actually two exceptions to that: a line that begins with '`#!`' doesn't imply a default of '`--cpp`', nor does a `#line` directive that is in the canonical form '`#line` *`digits`* `"`*`filename`*`"`' starting in column 1 and followed immediately by a newline. (Each space shown is a single space.) Such lines are instead transformed directly by the `setl` command into '`#` *`digits`* `"`*`filename`*`"`', just as `setlcpp` would do.
>
> At this time, `setl` has no option for spewing just the preprocessor output, although this can be retrieved from the `%SOURCE` section of the translator output (see [--compile], page 3), or generated directly using the `setlcpp` command.
>
> To apply the preprocessor, `setl` calls `setlcpp` with options '`-C`' and '`-lang-setl`' (see [setlcpp -lang-setl], page 11). The '`-lang-setl`' option is needed for correct processing of SETL string literals and comments. The '`-C`' (capital C) option means retain comments: this is used because there was once and may yet be again an escape convention where pseudo-code is

enclosed in '/* ... */'. In '-lang-setl' mode, '-C' also happens to cause SETL comments (not just C comments) to be retained in the preprocessor output.

'-I...'
'-D...'
'-U...' These options imply '--cpp' (see [--cpp], page 2), and are passed along to setlcpp in the order they occur.

There must be no space between the '-I', '-D', or '-U' and its sub-argument: '-I..' is good but '-I ..' fails with a message like 'setlcpp: Directory name missing after -I option'.

Directories listed in the SETL_INCLUDE_PATH environment variable (see [SETL_INCLUDE_PATH], page 12) will be searched *after* any specified via '-I' options when an #include directive is encountered. There are no predefined default search directories.

'--compile'
'-c' Runs only setltran, the GNU SETL translator (compiler). Produces (human-readable) GNU SETL virtual machine code on stdout.

'--translated'
'-t' Assumes that the input is GNU SETL virtual machine code, such as might have been produced by 'setl -c'.

'--font-hints'
Spews prettyprinting hints corresponding to the source input, and then exits successfully. Implies '--nocpp', though you can put a '--cpp' after the '--font-hints' option to override that and thereby get hints for the source as already preprocessed.

The hints are output as 3 integers: beginning offset, ending offset, and suggested font. There is one line of these per token of input. Comments count as whitespace. The offsets can be thought of as referring to the cracks between the characters, so if the first input character is a whole token by itself, its beginning and ending offsets are 0 and 1 respectively.

If the font codes are taken to mean roman for 1, italic for 2, and bold for 3, then predefined tokens of the SETL language will be in bold, literals in roman, and user identifiers in italics (though user-defined *operators*, i.e., those introduced by op or operator declarations, will be in bold). No font hints are given for comments, which probably look best in roman under this presentation scheme.

The '--font-hints' option is passed to setltran (see [setltran --font-hints], page 14).

Here is a little program called 'texinfo.setl' which produces Texinfo output (see the GNU *Texinfo* manual or http://www.gnu.org/software/texinfo/). It maps 2 to @emph and 3 to @strong, and leaves everything else in the default font. With its several single-letter variable names, it is perhaps not a splendid example of lucid SETL programming, but it has a couple of tuple formers that might amuse old fans of the World's Most Wonderful Programming Language:

```
pgmfile := command_line(1) ? 'texinfo.setl';
n := #'# 1 "' + #pgmfile + #'"' + 1;  -- ugh
p := fileno open ('setl --font-hints '+pgmfile, 'pipe-from');
hints := [[i-n,j-n,k] : doing reada(p,i,j,k); while not eof];
close(p);
s := getfile pgmfile;
m := 0;
putchar (''+/[at_sub s(m+1..i) + decorate (s(i+1..j), k) :
             [i,j,k] in hints step m := j; ]);
putchar (at_sub s(m+1..));

proc decorate (s, k);  -- decorate string s using font hint k
  case k
  when 2    => return '@emph{'+at_sub s+'}';
  when 3    => return '@strong{'+at_sub s+'}';
  otherwise => return at_sub s;
  end case;
end proc;

op at_sub (s);
  gsub (s, '@', '@@');  -- double any existing @ signs
  gsub (s, '{', '@{');  -- and take care of braces
  gsub (s, '}', '@}');
  return s;
end op;
```

Applied to itself (the default!), the above program's output is as follows. Note that this output looks sort of OK in HTML and in TEX-based renderings (DVI, whence PDF and PostScript), but is virtually illegible in an `info` reader. This program is of course a mere toy, however, and the gentle reader is referred to David Bacon's thesis, particularly Appendix A, for an example of what can be achieved with a much fussier and more comprehensive pretty-printer:

```
pgmfile := command_line(1) ? 'texinfo.setl';
n := #'# 1 "' + #pgmfile + #'"' + 1;  – ugh
p := fileno open ('setl –font-hints '+pgmfile, 'pipe-from');
hints := [[i-n,j-n,k] : doing reada(p,i,j,k); while not eof];
close(p);
s := getfile pgmfile;
m := 0;
putchar ('+/[at_sub s(m+1..i) + decorate (s(i+1..j), k) :
             [i,j,k] in hints step m := j; ]);
putchar (at_sub s(m+1..));

proc decorate (s, k);  – decorate string s using font hint k
  case k
  when 2    => return '@emph{'+at_sub s+'}';
  when 3    => return '@strong{'+at_sub s+'}';
  otherwise => return at_sub s;
  end case;
end proc;

op at_sub (s);
```

```
    gsub (s, '@', '@@');  – double any existing @ signs
    gsub (s, '{', '@{');  – and take care of braces
    gsub (s, '}', '@}');
    return s;
  end op;
```

Note how this program deals with the unpleasant fact that even programs that are not passed through `setlcpp` get a line of the form '`# 1 "filename"`' prepended on their way into `setltran`. A slightly simpler variation on this program is suggested in the corresponding `setltran` option advertisement (see [`setltran --font-hints`], page 14).

'`--keyword-case=upper|lower|any`'
'`--identifier-case=upper|lower|any|mixed`'

By default, the GNU SETL translator recognizes keywords and user identifiers case-insensitively, i.e., in `any` lettercase. Details on this and the other possibilities can be found with the corresponding `setltran` option description (see [`setltran` lettercase], page 14).

'`--maxmem=n`'

Limits the amount of memory that the GNU SETL virtual machine allows to be allocated for data.

The decimal number *n* may include a case-insensitive suffix `K` (1024), `M` (1024K), or `G` (1024M).

The default is unlimited, up to what the host system will bear. This default can be explicitly specified with '`--maxmem=0`'.

'`--restricted`'
'`-r`'

If '`--restricted`' (or equivalently '`-r`') is specified, the GNU SETL virtual machine disallows certain operations, such as file and socket operations, that can pose security risks.

Restricted mode is intended to let you run untrusted client programs. For example, you might wish to do this to let your students test and submit their SETL programs directly on and through your Web site. Dave's Famous Original SETL Server accepts programs through a web form and runs them in restricted mode.

This mode would also be suitable for a browser plugin that supports SETL "markup" (SETL program text embedded in Web pages).

To allow access to specific resources through the SETL `open` primitive even in restricted mode, use '`--allow-open`' and/or '`--allow-fd-open`' (see below).

'`--allow-open=file,mode ...`'
'`--allow-fd-open=fd,mode ...`'

These options are distinct though similar, and may be used multiple times. They drill little holes in the wall erected by "restricted" mode (see [`--restricted`], page 5), giving the SETL program access to just the particular resources you specify.

For example, you could give your students the time of day with '`--allow-open=localhost:daytime,socket`'.

Or, if you start their programs in an environment where file descriptor 4 is already open on some pipe, socket, or file you want them to be able to read from, then '`--allow-fd-open=4,r`' would do the trick.

Meaningful values of *`file`*, *`fd`*, and *`mode`* are those accepted by the SETL `open` primitive, as the above examples suggest.

Note that there should be no space around the comma in these options.

'`--help`'
'`-h`' Spews a command summary on stdout, and immediately exits with a success indication.

'`--version`'
Spews GNU SETL version information on stdout, and immediately exits with a success indication (0).

'`--verbose`'
'`-v`' Spews some tedious nonsense on stderr.

'`--debug`' In a normal build of `setl`, does nothing at run time. But if the preprocessor symbol `DEBUG_TRACE` was asserted, '`--debug`' causes instruction-by-instruction tracing of GNU SETL Virtual Machine execution, on stderr.

Regardless of `DEBUG_TRACE`, this option is passed to `setltran` (see [`setltran --debug`], page 14) if applicable.

A single hyphen is acceptable in place of the double hyphen in all the above options. Single-letter options only take a single hyphen, however. Also, single-letter options may not be "clustered": each option must be a separate argument, so '`setl -c -v`' wins but '`setl -cv`' loses.

Possibilities for the *`input`* argument to the `setl` command are tried in the following order:

'*`-fd`*' Program comes from the already open file descriptor *`fd`*, where *`fd`* is a decimal integer.

'`-`' Program comes from the standard input (stdin). This is the default if there are no other input arguments to `setl`.

'`|`*`command`*'
Program comes from the standard output of *`command`*.

'*`filename`*'
Program comes from the file *`filename`*.

'*`string`*' Program comes from the argument *`string`* itself.

Any run-time args after '`--`' (or alternatively '`-x`') appear as the tuple `command_line` in your SETL program.

The use of '`--`' is optional, but is generally recommended so that arguments intended for your program aren't mistaken as arguments to the `setl` command.

1.3 #! invocation and '-k'

On systems that support the "interpreter escape" convention in which any script beginning with the characters #! ("pound bang") is passed to the interpreter whose absolute pathname appears right after the #!, the setl command may be run indirectly to create SETL "scripts".

A special option, '-k', is needed to make this work properly. Let us begin with an example:

```
#! /usr/bin/setl -k
print (command_name, command_line);
```

If this script is put in /tmp/prtcmd and made executable, assuming setl is installed in /usr/bin, then the following invocation from a Bourne shell command line, and corresponding output, should be typical. The dollar sign is the usual default Bourne shell prompt for regular (non-privileged) users:

```
$ /tmp/prtcmd a2 "Hetu" "eh you" 57
prtcmd [a2 Hetu 'eh you' '57']
```

More generally, other setl options can be placed before the '-k' (none after). Note that the basename of the script becomes available to the script itself as the string command_name, and the arguments to the script turn up as the tuple of strings command_line.

Omitting option '-k' is possible, but denies setl the ability to tell that it is being called using the interpreter escape. If '-k' is omitted, there can be at most one token after the interpreter name, and it must be a valid setl option. The result of omitting '-k' is a script with a calling convention that resembles that of setl itself: if there are arguments, they must be valid setl options followed by any '--' (or equivalently '-x') argument, and then arguments intended for the script to process.

So except where a script writer deliberately wishes to provide user access to options controlling the underlying setl command (a questionable design principle), leaving out '-k' seems to be of little value.

Note also that without '-k', the value of command_name will be the name of the setl executable, rather than the name of the script.

Conversely, it is possible to use '-k' in a *direct* invocation of setl, in order to simulate what happens with the interpreter escape. Here is a useless but illustrative example:

```
$ cat /tmp/prtcmd.setl
print (command_name, command_line);
$ setl -k /tmp/prtcmd.setl a2 "Hetu" "b c" 57
prtcmd.setl [a2 Hetu 'b c' '57']
```

Notice how '-k' changes the calling convention of the setl command: after the '-k' must come a single filename and then the run-time arguments, since that is how arguments are presented in the interpreter escape case.

Finally, it is worth noting that you can easily embed multi-line SETL programs in shell scripts. This can be useful in order to avoid the need for absolute pathnames such as '/usr/bin/setl' at the top of '#!' SETL scripts.

For example, the Bourne shell script

```
#!/bin/sh
setl -3 3<<'!' -- "$@"
```

```
print ("Command args:", command_line);
print ("Please enter a number, string, set, or tuple:");
read (v);
print ("Thank you.  I now have", type v, "v =", v);
!
```

uses whatever version of the `setl` command comes first in the reigning `PATH`. The presence of `/bin/sh` is standard across POSIX systems, so `#!/bin/sh` is a very portable way to start a script.

In slightly more detail, the above script tells `setl` to read SETL source code from POSIX file descriptor 3, which is directed to the in-line "here" document that ends with a bang. The single quotes around the first bang indicate, in the bizarre and arcane language of the Bourne shell, that the SETL program text is to be taken literally, not subjected to parameter expansion, command substitution, or arithmetic expansion.

So, if the above script is executed or even just "sourced" by a Bourne-compatible shell, it will prompt on stdout and read from stdin, just as an equivalent '`#!`' SETL script would when executed, or as a free-standing SETL program run by the `setl` command would. Using this technique, it is easy to embed any number of SETL programs in a shell script.

Note, however, that if a program in-lined in that way has a syntax error or experiences a run-time error, the diagnostic will refer to a program named '`-3`', and a line number relative to where the program begins. A `#line` directive can be used to work around this problem. For example, if the above script is called `bah`, then the line

```
#line 4 "bah"
```

could be inserted as the first line of the program to ensure that diagnostics refer to `bah` and the correct line number of the script. Then if the user enters an invalid input, the diagnostic will point to line 6 of `bah`, the `read` statement.

That literal line number in the `#line` directive is a maintenance hazard, but if you are willing to assume a working `/bin/bash` and approve of the shell's making substitutions for `$...` and `` `...` `` things in your SETL program text, then a more maintenance-friendly way to embed it in `bah` would be:

```
#!/bin/bash
# Lines of shell script ...
setl -3 3<<! -- args to SETL program ...
#line $((LINENO+2)) "bah"
-- Lines of SETL program ...
!
# More lines of shell script ...
```

A very short SETL program can of course be entirely contained within a command-line argument; here is a functional equivalent to the `bah` script above:

```
#!/bin/sh
setl '
#line 4 "bah"
print ("Command args:", command_line);
print ("Please enter a number, string, set, or tuple:");
read (v);
print ("Thank you.  I now have", type v, "v =", v);
```

```
' -- "$@"
```

Note that special care must be taken of apostrophes in a program embedded in this last way (or of double quotes if those are used to enclose it), in order to keep the shell happy.

1.4 Environment variables

The environment variables to which the setl command is sensitive are as follows.

HOME
: This identifies the user's home directory, if any. It gives the default for the SETL chdir parameter.

SETLCPP
: If defined, gives the pathname of the GNU SETL preprocessor executable (usually named setlcpp). $SETLCPP is expanded by the shell, so it is possible to insert extra args into the preprocessor invocation this way.

SETLTRAN
: If defined, gives the pathname of the GNU SETL translator executable (usually named setltran). Expanded by the shell, like SETLCPP.

PATH
: If SETLCPP or SETLTRAN (as applicable) is undefined, but the argv[0] with which setl was invoked has a slash ('/') in it, then PATH (if defined) is extended to include the directory component of that ostensible pathname when the attempt to find and run setlcpp or setltran is made.

  For other commands launched by the SETL program, e.g., by system or filter, or by an open on a pipe or pump stream, the inherited PATH is used unmolested in locating the executable.

1.5 Signals

The signals that can be caught directly by the user's SETL program using open on a signal stream are those listed in Section "Signal streams" in the *GNU SETL Library Reference*.

1.6 Exit status

The setl command exits with a non-zero status in the event of an error. Specifically, if its invocation of setlcpp fails, setl returns its (error) status. Otherwise, if setltran fails, setl returns that status. Otherwise, if setl itself encounters a fatal (run-time) error, it issues a diagnostic and returns 1. But if a stop statement is executed, setl exits with the status given by the stop argument. That status defaults to 0, just as when the program flows through its last statement.

2 The setlcpp command

The setlcpp command is a modification of cpp, the GNU C PreProcessor. The main extension to GNU CPP is the provision of a '-lang-setl' option, which should normally be used when setlcpp is applied to SETL programs.

In the GNU SETL system, setlcpp is usually run automatically by the setl command (see [setl --cpp], page 2), and seldom directly from the interactive command line.

2.1 Examples

This is the output of the command 'setlcpp --help' on a typical system:

```
GNU SETL programming language preprocessor

Usage: setlcpp [OPTIONS] [INPUT [OUTPUT]]

  --help, -h       display this help on stdout and exit
  --version        display version info on stdout and exit
  -lang-setl       SETL lexical environment; implies -$
  CPP-OPTION       GNU CPP option

If INPUT is "-" or is not specified, standard input is used.
Otherwise, INPUT must name a readable file.  Similarly, OUTPUT
must name a writable file, or be "-" for the default stdout.

The "SETL_INCLUDE_PATH" environment variable, if -lang-setl is
specified, extends the list of directories given by -I options.
Directory names must be separated by a ":" character.

Other environment variables are as for GNU CPP version 2.7.2.1.

Note also the --[no]cpp option of the "setl" command, and see
http://setl.org for more documentation, source code, etc.

The command "info setlcpp" is also available on your system.

Please report bugs to bacon@cs.nyu.edu.
```

Now suppose the file 'main.setl' contains

```
-- This is a comment at the top of main.setl.
-- Let's now incorporate inc.setl:
#include "inc.setl"
print (corpor, version);
```

and the file 'inc.setl' contains

```
$ For nostalgic reasons, this is also a comment.
#define corpor "SETL, Inc."
```

```
const version = __VERSION__;
$ Here ends the included file.
```

Then the output of 'setlcpp -C -lang-setl main.setl' (which is how setl invokes setlcpp) is

```
# 1 "main.setl"
-- This is a comment at the top of main.setl.
-- Let's now incorporate inc.setl:
# 1 "inc.setl" 1
$ For nostalgic reasons, this is also a comment.

const version = "2.7.2.1";
$ Here ends the included file.
# 3 "main.setl" 2

print ("SETL, Inc.", version);
```

Note that a SETL const declaration may often serve as well as or better than a preprocessor #define. A preprocessor symbol, however, can be particularly useful for governing conditional source code inclusion via #if or #ifdef. Macros that take arguments have their uses too, though the user should always be aware of the literal expansion of arguments so as to be on guard against side-effects that result from multiple evaluations of an expression.

2.2 The setlcpp command and arguments

Here is the general form of the setlcpp command:

```
setlcpp [options] [input [output]]
```

The *options* include:

'--help'
'-h' Spews a command summary on stdout, and immediately exits with a success indication.

'--version'
Spews setlcpp version information on stdout, and immediately exits with a success indication.

'-lang-setl'
Assumes that the input is a SETL program, so that its lexical peculiarities can be accommodated. Otherwise, '-lang-c' (a *cpp-option*, see below) will be assumed.

The presence of '-lang-setl' also tells setlcpp to observe the SETL_INCLUDE_PATH environment variable (see [SETL_INCLUDE_PATH], page 12).

'*cpp-option*'
Any other argument beginning with a hyphen ('-') is interpreted as if by the GNU C Preprocessor (cpp), except that there are no predefined default "include" directories such as '/usr/include'.

As of this writing, setlcpp is based on the version of cpp corresponding to GCC 2.7.2.1. That original is bundled with the GNU SETL source distribution, including its Texinfo ('cpp.texi' and 'cpp.info*') documentation. For most purposes, however, you may find that the command 'info cpp' on your system or the on-line GNU CPP manual at http://www.gnu.org/software/gcc/onlinedocs/ gives adequate if somewhat anachronistic information. Otherwise, to get the version-specific truth, unpack 'cpp-2.7.2.1.tgz' and in the resulting subdirectory do this:

```
info -f ./cpp.info
```

If *input* is a hyphen ('-') or is not specified, setlcpp reads from standard input (stdin).

The *output* argument, which can only be present if *input* is present, must name a writeable file or be a hyphen representing standard output (stdout), the default.

Note that setlcpp is case-sensitive, regardless of any '--keyword-case' or '--identifier-case' options that might have been passed to a parent setl command.

Like cpp, the setlcpp command returns 0 to the operating system on success, non-zero on failure.

2.3 Environment variables

SETL_INCLUDE_PATH

If the '-lang-setl' option was given (see [setlcpp -lang-setl], page 11), SETL_INCLUDE_PATH extends the list of directories named in '-I' options, much as C_INCLUDE_PATH does in the '-lang-c' case. The standard cpp option '-nostdinc' will cause SETL_INCLUDE_PATH to be ignored, however.

For details on other environment variables, see the references cited under ['*cpp-option*'], page 11.

3 The setltran command

The setltran command takes SETL programs and compiles them into a simple assembly-like language that can be read by the GNU SETL virtual machine (setl) or by clients such as Zhiqing Liu's Lazy SETL Debugger.

In the GNU SETL system, setltran is usually run automatically by the setl command and seldom directly.

3.1 Examples

This is typical output of the command 'setltran --help':

```
GNU SETL programming language translator (compiler)

Usage: setltran [OPTIONS] [FILENAME | - | STRING]

  --help, -h      display this help on stdout and exit
  --version       display version info on stdout and exit
  --font-hints    emit source prettyprinting hints, period
  --verbose, -v   otiose sucrose on stderr
  --debug         trace parsing, etc. on stderr
  --keyword-case=any|upper|lower   ("stropping" convention) -
                  control SETL keyword recognition (default any)
  --identifier-case=any|upper|lower|mixed   control recognition
                  of user variable names (default any)

The setltran command reads from standard input by default or if "-"
is specified.  Otherwise, if FILENAME names a readable file, it
reads from there.  Failing that, it reads directly from STRING.

When the translator is invoked by a command like "setl -c ...",
the preprocessor (setlcpp) is applied first if necessary.

See http://setl.org for more documentation, source code, etc.

The command "info setltran" is also available on your system.

Please report bugs to bacon@cs.nyu.edu.
```

This is the output (on stdout) of the command 'setltran "print(57);"':

```
# This is code for the GNU SETL Virtual Machine.
%SOURCE
print(57);
     1  print(57);
%CODE
#       print(57);
```

```
        0          mainproc        U__MAIN
        0          call    U__unnamed_SETL_program 0       >-
        0          copy    <I_0    >STATUS
        0          stop    <STATUS
        0          proc    U__unnamed_SETL_program
        6          copy    <[]     >T_1
        6          extend_tuple    <I_57   <>T_1
        6          truncate        <>T_1
        6          scall   S_PRINT 1       <T_1    >-
        6          copy    <I_0    >STATUS
        6          stop    <STATUS
    %EXECUTE
```

For more information on GNU SETL virtual machine code, see the *GNU SETL Implementation Notes* [stub].

3.2 The `setltran` command and arguments

Here is the general form of the `setltran` command:

`setltran [`*`options`*`] [`*`input`*`]`

The options include:

'`--help`'
'`-h`' Spews a command summary on stdout, and immediately exits with a success indication.

'`--version`'
Spews `setltran` version information on stdout, and immediately exits with a success indication.

'`--font-hints`'
Spews prettyprinting hints corresponding to the source input, and then exits (successfully).

It is left as an exercise to the reader to simplify the '`texinfo.setl`' program in the description of the `setl` command's corresponding option (see [`setl --font-hints`], page 3). Hint: by invoking `setltran` as the subprocess instead of `setl`, you can eliminate the offset variable, ***`n`***. For extra credit, state why.

'`--verbose`'
'`-v`' On stderr, reports each of the major phases of activity, i.e., lexical analysis, parsing, semantic analysis, etc.

'`--debug`' On stderr, traces the gory details of shift-reduce parsing and dumps some tables. The author really prefers recursive descent parsers: `setltran` was an aberration.

This option also causes files '`parse.tree`' and '`flattened.tree`' to be created in the current working directory.

‘`--keyword-case=upper|lower|any`’
‘`--identifier-case=upper|lower|any|mixed`’

By default, `setltran` recognizes keywords and user-introduced identifiers case-insensitively, i.e., in `any` lettercase.

However, it can be useful to restrict the recognition for stylistic or maintenance reasons.

For example, since new keywords sometimes enter the language, and “customized” implementations can add many more, one might adopt a convention of `--keyword-case=upper` to ensure that only uppercase identifiers are recognized as keywords, no matter what new keywords might later be introduced.

C/C++ and Java programmers might like a combination of `--keyword-case=lower` and `--identifier-case=mixed` for a familiar case-sensitive environment.

The `setl` command passes these lettercase options through to `setltran`.

The `setltran` command takes up to one *`input`* argument in addition to any options specified. It must be the name of a readable file, or a single hyphen (‘`-`’) meaning standard input (stdin), or a string containing an entire SETL program. If there is no *`input`* argument, stdin is assumed.

Finally, `setltran` exits with a code of 0 on success, or a higher number on failure.

Index

#

`#!` invocation, '`-k`' option 7
`#define` 3
`#if`, `#ifdef` 2
`#include` 3
`#line` directive 2, 8
`#undef` 3

-

'`-`' (read from stdin) 6, 15
'`-`' (stdin or stdout, `setlcpp`) 12
'`--`' (begin run-time args) 6
'`--allow-fd-open`' option 5
'`--allow-open`' option 5
'`--compile`' option 3
'`--cpp`' option 2
'`--debug`' option 6, 14
'`--font-hints`' option 3, 14
'`--help`' option 6, 11, 14
'`--identifier-case`' option 5, 15
'`--keyword-case`' option 5, 15
'`--maxmem`' option 5
'`--nocpp`' option 2
'`--restricted`' option 5
'`--translated`' option 3
'`--verbose`' option 6, 14
'`--version`' option 6, 11, 14
'`-c`' option 3
'`-D`' option 3
'`-h`' option 6, 11, 14
'`-I`' option 3, 12
'`-k`' option 7
'`-lang-setl`' option 11, 12
'`-nostdinc`' option 12
'`-r`' option 5
'`-t`' option 3
'`-U`' option 3
'`-v`' option 6, 14
'`-x`' option 6

/

`/bin/bash` 8
`/bin/sh` 7

A

allow selective file and fd opening 5
'`any`' case option 15
args, run-time, after '`--`' 6

C

`C_INCLUDE_PATH` 12
case, keyword and identifier 15
client programs, access to resources 5
client programs, untrusted, restricted mode 5
command, reading program from output of 6
command-line argument, reading program from .. 6
`command_line` 7
`command_line` in relation to '`--`' 6
`command_name` 7
compile-only mode 3
compiler 3, 13
`cpp` (GNU C Preprocessor) 11

E

environment variables, `setl` command 9
environment variables, `setlcpp` command 12
examples, `setl` 1
examples, `setlcpp` 10
examples, `setltran` 13
executables, search path for 9
exit status, `setl` command 9
exit status, `setlcpp` command 12
exit status, `setltran` command 15

F

file descriptor, reading program from 6
file, reading program from 6
font hints 3, 14

G

GNU C Preprocessor (`cpp`) 11

H

help, `setl` 6
help, `setlcpp` 11
help, `setltran` 14
hints, font 3
`HOME` environment variable 9

I

identifier case, recognizing 15
interpreter escape, '`-k`' option 7

K

keyword case, recognizing 15

L

Lazy SETL Debugger 13
lettercase 5, 15
limit, memory 5
Liu, Zhiqing 13
'`lower`' case option 15

M

macro 2
memory limit 5
'`mixed`' case option 15

O

`open`, selectively allowing 5

P

parameters, run-time, after '`--`' 6
`PATH` environment variable 9
pound bang (`#!`), '`-k`' option 7
preprocessor 2, 10
prettyprinting hints 3, 14

R

reading program from command output 6
reading program from command-line argument .. 6
reading program from file 6
reading program from file descriptor 6
reading program from stdin 6
reading program from string 6
recognizing keywords, identifiers 15
restricted mode 5
run-only mode 3
run-time args (parameters) after '`--`' 6

S

search path for executables 9
security, allowing selective access to resources ... 5
security, restricted mode 5
SETL script, '`-k`' option 7
`SETL_INCLUDE_PATH` 3, 12
`SETLCPP` environment variable 9
`setlcpp` location 9
`SETLTRAN` environment variable 9
`setltran` location 9
shebang (`#!`), '`-k`' option 7
shell script, embedding SETL programs in 7
signals, `setl` command 9
stdin, reading program from 6
string, reading program from 6

T

Texinfo 1, 3, 11
translator 3, 13

U

untrusted client programs, access to resources ... 5
untrusted client programs, restricted mode 5
'`upper`' case option 15

V

verbose mode 6, 14
version, `setl` 6
version, `setlcpp` 11
version, `setltran` 14
virtual machine 1
virtual machine code 3, 14