GNU SETL User Guide

Edition 3.5.8, for GNU SETL Version 3.5.8
Updated 10 December 2022

by dB

Table of Contents

The setl command

The `setl` command is the primary interface to the GNU SETL system. In typical use, it preprocesses SETL programs using `setlcpp` and compiles them with `setltran` into GNU SETL Virtual Machine code, which it then executes.

Depending on the 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, receive timer events, listen on server ports, and open and use files, network connections, and existing file descriptors.

Examples

This is the output of the command '`setl --help`':

```
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       emit 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=WHAT,HOW ...   restriction exemptions for open()
  --allow-fd-open=FD,HOW ...   exemptions for open() over fd
  --allow-mkstemp=TEMPLATE ...   exemptions for mkstemp() calls
  --allow-filter=COMMAND ...   exemptions for filter() calls
  --allow-system=COMMAND ...   exemptions for system() calls
  --setlcpp=COMMAND   specify preprocessor command
  --setltran=COMMAND  specify translator command
  --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
  --abort-on-error    dump core for SETL-level error

  -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)
```

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

If the Texinfo documentation is installed, "info setl" may work.
PDF and HTML docs are usually under share/doc/setl/ somewhere.

See setl.org for more documentation, source code, etc.

Please report bugs to David.Bacon@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']
```

The `setl` command and arguments

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

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

The *`options`* include:

`--[no]cpp`

By default, the GNU SETL Preprocessor, setlcpp, an adaptation of GNU CPP (the GNU C Preprocessor), is applied if the input program appears to require it. Use `--cpp` or `--nocpp` to make an explicit choice. Options `-I...`, `-D...`, and `-U...`, 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 appears to have possible `setlcpp` directives. Because a false positive is usually 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 some 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.

Also, the presence of identifiers starting with a double underscore, such as `__VERSION__`, `__FILE__`, or `__LINE__`, imply `--cpp`, and are transformed by `setlcpp` appropriately if recognized. Note that no SETL variable identifier can begin with an underscore, so such symbols can only be preprocessor symbols (like those introduced via '`#define`' or a `-D...` option).

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`) or generated directly using the `setlcpp` command.

To apply the preprocessor, `setl` calls `setlcpp` with options `-C` and `-lang-setl`. 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` 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 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` (`setltran --font-hints`).

Here is a little program called `texinfo.setl` which produces Texinfo output (see the GNU *Texinfo* manual or `https://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 dB's thesis, Appendix A, for an example of what can be done 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 description (`setltran --font-hints`).

`--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 descriptions (`setltran --keyword-case` and `setltran --identifier-case`).

`--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. For specifics, see Section "Restricted Mode" in the *GNU SETL Library Reference*.

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 even in restricted mode, use as many `--allow-...` options as required.

`--allow-open=`*`what,how`* ...
`--allow-fd-open=`*`fd,how`* ...
`--allow-mkstemp=`*`template`* ...
`--allow-filter=`*`command`* ...
`--allow-system=`*`command`* ...

These options, which may be used multiple times, drill little holes in the firewall erected by the `--restricted` option, giving the SETL program access to particular resources specified at `setl` invocation time.

For example, you can give your students the time of day with '`--allow-open=profhost:daytime,tcp-client`'.

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.

The arg '`--allow-mkstemp=/tmp/homework-1XXXXXX`' allows the SETL `mkstemp` primitive to be called with the given template, for the safe use of temporary scratch files in restricted mode.

Likewise, '`--allow-filter=fmt`' lets the SETL program apply `filter` to the commmand `fmt`, and '`--allow-system="mail prof </tmp/summary$(uid)"`' lets the program call `system` with a very particular `mail` command.

Note that commands, filenames, and templates in `--allow-...` args will require appropriate quoting to deal with internal spaces and other special characters when a standard Bourne-type shell is used to invoke `setl`, as that latter example illustrates.

Meaningful values of *`what`*, *`fd`*, and *`how`* are those accepted by the SETL `open` primitive, except that a `tuple` argument to `open` must be represented as a pair of strings separated by a colon in `--allow-open` options. Specify integer values as strings of decimal digits.

There should be no space around the comma that separates *`what`* or *`fd`* from *`how`*.

Timer streams are always allowed to be opened, without the need to give an `--allow-open` option for them.

The *`what`* part of an `--allow-open` argument must be matched exactly in the SETL program's `open` call (or equivalent auto-open), with these exceptions: (1) the names of signal-catching, signal-ignoring, and signal-defaulting streams need only be equivalent according to the usual `open` convention; and (2) when *`what`* is a network (host:service) spec, the matching is case-insensitive.

The case-sensitive matching for commands and filenames is the safest way to treat an `--allow-open` security exemption, even though `open` itself may behave case-insensitively on some combinations of OS and filesystem.

`--setlcpp=`*`command`*

This specifies a preprocessor command to be used in place of the default `setlcpp`. The default is that if `setl` appears to have been invoked using a specific pathname (i.e., there is a directory separator character in `argv[0]` at the C level), then `setlcpp` is sought in the same directory as `setl` was ostensibly found in. Otherwise, given no directory separator character, the `PATH` environment variable is searched in the usual POSIX way for a directory containing an executable `setlcpp`.

The ***`command`*** in the `--setlcpp` option is in fact taken as the initial substring of a command to be passed, unquoted, with appended args such as `-D...` and `-I...`, quoted, to the POSIX standard (Bourne-compatible) shell. Thus `PATH` is also consulted if the specified ***`command`*** has no directory separator character in its first token.

`--setltran=`*`command`*
: This specifies a translator command to be used in place of the default `setltran`. The default that if `setl` appears to have been invoked using a specific pathname (i.e., there is a directory separator character in `argv[0]` at the C level), then `setltran` is sought in the same directory as `setl` was ostensibly found in. Otherwise, given no directory separator character, the `PATH` environment variable is searched in the usual POSIX way for a directory containing an executable `setltran`.

The ***`command`*** in the `--setltran` option is in fact taken as the initial substring of a command to be passed, unquoted, with appended args such as `--verbose`, to the POSIX standard (Bourne-compatible) shell. Thus `PATH` is also consulted if the specified ***`command`*** has no directory separator character in its first token.

`--help`
`-h` Spews a command summary on stdout, and exits successfully.

`--version`
Spews GNU SETL version information on stdout, and exits successfully.

`--verbose`
`-v` Spews some garbage on stderr during execution for the amusement of nerds.

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

Regardless of `DEBUG_TRACE`, this option is passed to `setltran` (`setltran --debug`).

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.

#! invocation

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

Here is an example:

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

If that script is put in `/tmp/prtcmd` and made executable, and if `setl` (together with `setlcpp` and `setltran`) is installed in `/usr/bin`, then the shell command

```
/tmp/prtcmd a2 'Hetu' 'eh you' 57
```

will give the output

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

Note that the pathname of the script is available to the program as the string `command_name`, and the arguments to the script as the tuple of strings `command_line`.

Another possibility is to begin the shell script as follows when you don't wish to specify an absolute pathname for the `setl` command but would rather have it found in the user's `PATH`, and don't need to pass any options to `setl`:

```
#! /usr/bin/env setl
```

Multi-line SETL programs can also be embedded in shell scripts. Example:

```
#! /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);
!
```

The above script tells `setl` to read 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 an execution 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 `bach`, then the line

```
#line 4 "bach"
```

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

That literal line number in the #line directive is obviously a maintenance hazard, but if you are willing to assume a working /bin/bash, and approve of the shell making "here-document" substitutions for sequences such as $... in your SETL code, then another way to embed it in bach is:

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

Finally, a very short SETL program can be entirely contained within a command-line argument; here is a functional equivalent to the bach scripts above:

```
#! /bin/sh
setl '
#line 4 "bach"
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.

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.

PATH
: For commands launched by your SETL program, e.g., by system or filter, or by an open on a pipe or pump stream, the PATH environment variable is used in locating the executable. PATH is also used in searching for the setlcpp and setltran executables when setl itself appears to have been found in a directory listed in PATH (i.e., when there is no directory separator character in argv[0] at the C level) and where this search is not overridden by a --setlcpp or --setltran option.

SETL_LINEBUF_STDERR
: By default, characters on the standard output stream (stderr) are flushed (written out) as soon as possible; i.e., the stream is *unbuffered* (see Section "Buffering" in the *GNU SETL Library Reference*). But if SETL_LINEBUF_STDERR is set (to anything, even the null string), then stderr is *line buffered*, meaning

that characters may not be written out until the next newline is written to stderr by the SETL program.

This can be convenient when a bunch of different processes all want to issue diagnostics to the terminal at the same time, as it greatly reduces the likelihood that those messages will be intermingled on a character-by-character basis. In a production-level set of related processes, of course, it is probably better to redirect everyone's stderr to a common server that respects newlines and can perform additional functions such as keeping a log, presenting a highlighted real-time display, etc.

Signals

For the signals that can be caught directly by the user's SETL program using `open` on a signal stream, see Section "Signal streams" in the *GNU SETL Library Reference*.

Exit status

The `setl` command exits with a non-zero status in the event of an error. Specifically, if the invocation of `setlcpp` fails, `setl` returns its (error) status. Otherwise, if `setltran` fails, `setl` returns that status. (If the failure of `setlcpp` or `setltran` is due to termination by a signal, the status will be 128 plus the signal number, in mimicry of the standard shell convention.) Otherwise, if `setl` itself encounters an unrecoverable 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.

Note that although `stop` accepts any integer small enough to fit into a C `int`, it is returned modulo 256 to the invoker of the `setl` command.

If you want your program's exit status to be that of the last subcommand it waited for, and mimic the shell in the case of abnormal termination by signal, you could use this little horror:

```
stop if status >= 0 then status else 128 + abs status end;
```

Otherwise, if you only care that your exit status be zero or nonzero according as the last subcommand succeeded or failed, or you know that the last subcommand was actually being managed by an enclosing shell (the usual case, unless you begin the subcommand with the word `exec`), or even if you just don't mind having the exit values associated with signal-triggered terminations a little weirdly mapped (the other exit codes will come through fine), you can generally get away with the much simpler

```
stop status;
```

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 and seldom directly from the interactive command line.

Examples

This is the output of the command 'setlcpp --help':

```
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.

If the Texinfo documentation is installed, "info setlcpp" may work.
PDF and HTML docs are usually under share/doc/setl/ somewhere.

See setl.org for more documentation, source code, etc.

Please report bugs to David.Bacon@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.

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 exits successfully.

`--version`
Spews `setlcpp` version information on stdout, and exits successfully.

`-lang-setl`
Assumes that the input is a SETL program, so that its lexical peculiarities can be accommodated. Otherwise, `-lang-c`, which was once a standard `cpp` option, will be assumed.

The `-lang-setl` option also makes `setlcpp` recognize the `SETL_INCLUDE_PATH` environment variable.

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://gcc.gnu.org/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 writable file or be a hyphen representing standard output (stdout), the default.

Note that setlcpp is case-sensitive despite 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.

Environment variables

SETL_INCLUDE_PATH

If the setlcpp -lang-setl option was given, then 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 12.

The setltran command

The `setltran` command takes SETL programs and compiles them into a simple assembly-like language that the `setl` command can interpret as GNU SETL Virtual Machine code.

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

Examples

This is the 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.

If the Texinfo documentation is installed, "info setltran" may work.
PDF and HTML docs are usually under share/doc/setl/ somewhere.

See setl.org for more documentation, source code, etc.

Please report bugs to David.Bacon@nyu.edu.
```

This is the output (on stdout) of the command '`setltran "print(57);"`', with tab stops every 8 columns (tabs separate opcodes and operands):

```
# 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          copy    <I_0    >STATUS
     0          stop    <STATUS
     0          end     U__MAIN
     0          proc    U__unnamed_SETL_program >RET
     6          scall   S_PRINT <I_57   >-
     6          copy    <I_0    >STATUS
     6          stop    <STATUS
     6          end     U__unnamed_SETL_program
%EXECUTE
```

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

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 exits successfully.

`--version`
Spews `setltran` version information on stdout, and exits successfully.

`--font-hints`
Spews prettyprinting hints corresponding to the source input.

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 (`setl --font-hints`). 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 processing, such as 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` is 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.

That happens to correspond to the ancient and venerable Algol 68 *upper* stropping convention. *Point* stropping, incidentally, was quite dreadful: keywords had to be denoted by leading dots. Fortunately for Algol 68 programmers, there was also *res* (standing for *reserved word*) stropping, which imposed no restriction on the lettercase of keywords. That corresponds to the GNU SETL default.

A combination of `--keyword-case=lower` and `--identifier-case=mixed` gives a case-sensitive convention resembling that of C/C++ and Java.

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 (the default), or a string containing an entire SETL program.

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

Index