

GNU SETL Om

On the World's Most Wonderful Programming Language
Edition 8.13.22, for GNU SETL Version 8.13.22
Updated 10 December 2022

by dB

Table of Contents

| | |
|--|-----------|
| Introduction | 1 |
| General characteristics of SETL | 1 |
| History | 1 |
| To the Student of Programming Languages | 4 |
| GNU SETL | 7 |
| SETL Variations | 8 |
| Links | 9 |
| Other GNU SETL Manuals | 9 |
| External Links | 9 |
| Administrivia | 11 |
| Copying GNU SETL | 11 |
| Reporting Bugs | 11 |
| Index | 12 |

Introduction

General characteristics of SETL

The SETL programming language describes sets, and in particular maps, in notations similar to those of mathematical set theory. For example,

$$\{1..n\}$$

is the set of integers 1 through n , and

$$\{p \text{ in } \{2..n\} \mid \text{forall } i \text{ in } \{2..p-1\} \mid p \bmod i \neq 0\}$$

is the set of primes through n , which can be read out loud as “*the set of all p in $\{2..n\}$ such that for all i in $\{2..p-1\}$, the remainder of p divided by i is non-zero*”, meaning each p is prime that has no divisors in $\{2..p-1\}$.

The vertical bar, for *such that*, takes an iterator on the left and a boolean condition on the right, generally involving the iteration variables; **forall** exercises another iterator and returns **true** if its such-that condition is true on all iterations; \neq means inequality.

Logically enough, the following is the same set of primes as the above:

$$\{p \text{ in } \{2..n\} \mid \text{notexists } i \text{ in } \{2..p-1\} \mid p \bmod i = 0\}$$

Tuples, like sets, are central enough to the language to get their own syntax. The value

$$["age", 21]$$

is a 2-tuple or *ordered pair*. There is no restriction on the types of the values within sets or tuples, and no predefined restriction on the degree to which they can be nested.

A set consisting entirely of ordered pairs is a map. For a map f having the value

$$\{["name", "Jack"], ["age", 21]\}$$

the value of $f("age")$ (or $f.age$) is thus 21, and $f\{"age"\}$ is the set of *all* the values corresponding to "age". In this instance $f\{"age"\}$ is just the singleton set $\{21\}$, but this notation is useful when f could be a *multi-map*, taking some domain points to more than one range value.

SETL also has the customary control structures of a procedural programming language, such as **if**, **case**, **while**, and **for**. Loops can often be replaced by set-forming and tuple-forming expressions for a more concise and direct functional style, too.

For more on iterators, maps, and the *strict value semantics* of SETL, see [\[To the Student of Programming Languages\]](#), page 4.

History

SETL began as a tool for the high-level expression of complex algorithms. It soon found a role in *rapid software prototyping*, as was demonstrated by the NYU Ada/Ed project, where it was used to implement the first validated Ada 83 compiler and run-time system. The prototyping was rapid, but the CIMS SETL implementation was not; the team made *slow is beautiful* a mantra and point of pride.

But as Robert Dewar liked to point out, the readability of a program is more important than how quickly it can be written. Jack Schwartz's original monograph, *Set Theory as a*

Language for Program Specification and Programming (1970), made clarity of expression a primary goal of SETL from the outset:

In the present paper we will outline a new programming language, designated as SETL, whose essential features are taken from the mathematical theory of sets. SETL will have a precisely defined formal syntax as well as a semantic interpretation to be described in detail; thus it will permit one to write programs for compilation and execution. It may be remarked in favor of SETL that the mathematical experience of the past half-century, and especially that gathered by mathematical logicians pursuing foundational studies, reveals the theory of sets to incorporate a very powerful language in terms of which the whole structure of mathematics can rapidly be built up from elementary foundations. By applying SETL to the specification of a number of fairly complex algorithms taken from various parts of compiler theory, we shall see that it inherits these same advantages from the general set theory upon which it is modeled. It may also be noted that, perhaps partly because of its classical familiarity, the mathematical set-notion provides a comfortable framework, that is, requiring the imposition of relatively few artificial constructions upon the basic skeleton of an analysis. We shall see that SETL inherits this advantage also, so that it will allow us to describe algorithms precisely but with relatively few of those superimposed conventions which make programs artificial, lengthy, and hard to read.

Sets and maps abound in abstract, human-oriented presentations of algorithms, and those can often be turned into concrete SETL programs with remarkably little change.

In its focus on clarity of expression, SETL also seeks to help programmers avoid the trap of premature optimization. Quoting Schwartz again, this time from *On Programming* (1973):

On the one hand, programming is concerned with the specification of algorithmic processes in a form ultimately machinable. On the other, mathematics describes some of these same processes, or in some cases merely their results, almost always in a much more succinct form, yet in a form whose precision all will admit. Comparing the two, one gets a very strong even if initially confused impression that programming is somehow more difficult than it should be. Why is this? That is, why must there be so large a gap between a logically precise specification of an object to be constructed and a programming language account of a method for its construction? The core of the answer may be given in a single word: efficiency. However, as we shall see, we will want to take this word in a rather different sense than that which ordinarily preoccupies programmers. More specifically, the implicit dictions used in the language of mathematics, which dictions give this language much of its power, often imply searches over infinite or at any rate very large sets. Programming algorithms realizing these same constructions must of necessity be equivalent procedures devised so as to cut down on the ranges that will be searched to find the objects one is looking for. In this sense, one may say that *programming is optimization* and that mathematics is what programming becomes when we forget optimization and *program in the manner appropriate for an infinitely fast machine with infinite amounts of memory*. At the most fundamental level, it is the mass of optimizations with which it is burdened that makes programming so cumbersome a process, and it is

the sluggishness of this process that is the principal obstacle to the development of the computer art.

For a further survey of early SETL history, including numerous bibliographic references, please see [dB's thesis](#).

Also, [Paul McJones](#) has compiled a more complete account of SETL history in the [SETL Historical Sources Archive](#) at the [Computer History Museum](#).

To the Student of Programming Languages

There are no pointers (no references, no aliases) in SETL, and it is natural to wonder how such a language could be convenient or even widely useful.

The answer is the maps. Where you might be tempted to think pointers are necessary, such as in a linked data structure, it is usually better to make the nodes the range elements of a map over a more interesting domain than memory pointers, and let the node references be the keys of that map.

This pointer-free aspect of SETL is sometimes called its strict *value semantics*. Conceptually, every assignment or parameter pass means copying. For a set or tuple, copying applies recursively.

Programming without pointers seems to be much simpler than with them, as all the hazards of aliasing and of “dangling” pointers disappear. But what about the cost of all that copying?

As Annie Liu observes in [some notes on files received from Jack Schwartz](#), the value semantics makes powerful optimization techniques such as Paige’s finite differencing much easier to implement than when aliases have to be considered.

Alas, GNU SETL does not employ those advanced techniques. It copies in many places where the copying could in principle be optimized or differenced away. It is surprising how seldom that matters in the typical data processing setting, however. Sets and in particular maps are efficiently implemented, which helps.

Maps are central to SETL. Here is how a map `count` might be built up, using a loop to tally input word frequencies:

```
count := {};           -- empty map
for word in split(getfile stdin) loop -- whitespace-delimited words
    count(word) += 1;   -- init to 1, or add 1
end loop;
```

We can now print a crude histogram from the `count` map:

```
for [word, freq] in count loop -- loop through the map
    print(freq * "*", word);    -- print freq stars and the word
end loop;
```

In that way of iterating over the map, successive ordered pairs are broken out from `count` and assigned to `[word, freq]`, meaning `word` gets the string and `freq` gets the associated tally on each iteration.

EXERCISE 1. A set-building notation was introduced in the prime-numbers example (see [\[primes\]](#), page 1). So now, for any given map `m`, what does

```
{[y, x] : [x, y] in m}
```

represent?

You may guess that it is the inverse of `m` (which it is), but is that colon a misprinted vertical bar?

Actually, it is not. The general form of the inside of a *set former* such as the above is:

```
expression : iterator | condition
```

The *expression* characterizes each member of the set being formed, and is evaluated for each iteration of the *iterator* for which the *condition* is true. *Bound* variables (like `word`,

or like *x* and *y*) are assigned values on each iteration, and will often appear in the *condition* and *expression*.

The *iterator* itself can actually be a series of chained iterators separated by commas, where the iterators to the right loop within those to the left, and can therefore refer to variables bound by them.

A *tuple former* is just like a set former but uses square brackets (`[]`) instead of curly braces (`{}`). It constructs a tuple with elements in the order produced by iteration.

The general form shown above has two main degenerate forms. You can omit the *condition*, leaving

expression : *iterator*

as in the map-inverse example above, in which case the condition defaults to *true*. Or you can omit the *expression*, leaving

iterator | *condition*

as in the prime-numbers example (see [\[primes\]](#), page 1). In that example, the *expression* defaults to *p*, the expression to the left of the first *in*.

The other major users of iterators in SETL, besides the loops and the set and tuple formers, are the quantified expressions, which have *forall*, *exists*, or *notexists* followed by *iterator* | *condition*. For example,

forall *i* in [2..10] | 11 mod *i* /= 0

is *true*, because 11 is in fact prime.

Let us now examine iterators in a little more detail. The most common kind has the general form

lhs in *s*

where the *lhs* is anything that can be assigned to, and *s* is an already existing set, string, or tuple. For a “structured” *lhs* such as [*x*, *y*] or [[*x*, *y*], *z*], every member of *s* must be a tuple that is structurally compatible with *lhs*, by the same rule that governs parallel assignments such as

[*a*, *b*] := [*b*, *a*]

which swaps the values of variables *a* and *b*.

When used in a set or tuple former, the *lhs* part of the above *lhs* in *s* iterator form serves also as the default expression when the *expression* is omitted.

For SETL beginners with a background in formal set theory, the resemblance between iterator-based SETL set formers and abstract mathematical set builders provides a dual view of the set as an object to be built up by computation on the one hand and as an intensionally defined comprehension on the other. In both cases, the *expression* gives the “shape” of the elements. There is a similar duality between the quantified expressions of SETL (*forall*, *exists*, *notexists*) and those of mathematical logic.

For SETL beginners *without* a background in set theory or logic, but with some familiarity with other programming languages, the iterators in formers and quantified expressions may be the least obvious aspect of SETL. The thing to remember is that they represent loops. Where you see a set former that looks as if it contains undefined variables, look to see if those variables are actually the bound variables of an iterator that is just after the colon

and/or just before the vertical bar. An iterator will always have either the keyword `in` or an equals sign (`=`) in it.

An additional role for the `in` keyword is as the name of a binary operator;

```
x in s
```

is a boolean-valued membership test for whether the value `x` occurs in `s`.

EXERCISE 2. Given sets `s1` and `s2`, identify the iterator and the boolean expression in

```
{x in s1 | x in s2}
```

EXERCISE 3. Given the same sets again, state whether that set is the same as

```
{x in s2 | x in s1}
```

The answer to the latter is indeed yes, these both represent the intersection of `s1` and `s2` (which could be written in SETL more simply as `s1 * s2`). But in the absence of some fairly sophisticated code optimization, they get there by different means: if `s1` has a huge number of members, and `s2` very few, it will take much longer to do it the first way (iterating through `s1` and testing each member for membership in `s2`) than the second.

The order in which members are selected during iteration over a set in SETL is left up to the implementation. It is *arbitrary*. Similarly, the SETL `arb` (“choice”) operator selects a set member arbitrarily, and the SETL `from` statement selects and removes an arbitrary member from a set. Programmers should treat this arbitrariness as nondeterministic, but *not random*. “Random” implies some kind of stochastic process in the selection, and SETL has a separate operator to approximate that, called `random`.

SETL has further iterator forms such as

```
y = f(x)
```

for iterating over a single-valued map `f` while assigning successive corresponding domain and range values to `x` and `y` respectively. For the single-valued map case, this iteration is equivalent to the form

```
[x, y] in f
```

so we could tighten up the loop header in the word-counting example to read

```
for freq = count(word) loop
```

in order to document and check that the map `count` is expected to be single-valued. If `count` were any other kind of set, such as a multi-map or a set containing something other than ordered pairs, a run-time error would result.

Once again, we see a strong syntactic resemblance between an iterator and a boolean expression; and “`freq = count(word)`” is certainly true within the body of the above loop. That functional-style iterator form is also applicable when `f` is a string or tuple, in which case `x` successively takes on the values 1 through the length of the string or tuple, as `y` gets assigned the corresponding characters of the string or members of the tuple. This is also consistent with the notation for element access (“subscripting”) for these types.

Multi-map iteration is indicated by braces rather than parentheses, as in multi-map selection:

```
ys = f{x}      -- ys gets range subset for each x
```

These iterator shorthands also work like their corresponding selection operations:

```
z = f(x, y)    -- means z = f([x, y])
zs = f{x, y}   -- means zs = f{[x, y]}
```


GNU SETL

GNU SETL is an implementation of SETL, with a few extensions. The goal was always a `setl` command that would play well in a Unix-like environment, allowing it to be used as easily as say `awk`, `sed`, or `grep`. Even with an external interface consisting of little more than basic I/O, `setl` did indeed prove to be a valuable tool in a variety of roles, from combinatorial experiments to routine scientific data processing. However, there came a time when it seemed worthwhile to add new facilities for working more directly with processes, signals, timers, and sockets.

SETL is something of a natural for event-driven and distributed systems. Passing SETL maps etc. between processes is particularly easy, as most values can be serialized for transport by a simple `write` and deserialized by a simple `read`.

Neither the SETL language nor the GNU SETL implementation is well suited to “programming in the large”. But Unix has taught us how useful it can be to chain processes into pipelines in everyday data processing, and I personally have had great fun prototyping event-driven systems as trees of processes, all on the same general pattern as advertised in the case study in [dB’s thesis](#). The SETL programs in those systems range from tiny (buffers, dispatchers, multiplexers, etc.) to medium-sized. The decentish high-level string handling in GNU SETL, and its rich repertoire of process management, let it deal pretty competently with external programs, especially the many text-based utilities in the Unix/POSIX canon. Another reusable text-based pattern is seen in practice where `wish` interpreter is invoked as a subprocess and fed Tcl/Tk commands to build a GUI based on traversal of a nested SETL tuple that specifies the elements and layout in a rather declarative manner. Events from the widgets come back as text, and updates are sent as further Tcl/Tk commands.

Here is a summary of GNU SETL’s main extensions to SETL:

- a POSIX-based interface to networks, signals, timers, and processes;
- substring extraction and substitution by regular expression;
- support for SETL2 control structure syntax and packages;
- a dot notation `f.x` as syntactic sugar for `f("x")`, letting a map `f` act as a record with a field `x` that may come and go dynamically;
- procedure references and indirect calls;
- a notation that allows one variadic procedure to call another with trailing args presented as if they had come from the caller of the first procedure, for use in implementing things like logging functions where you want to pass the top-level caller’s args to `print` after some context args such as a timestamp.

For a list of GNU SETL functions, operators, and special values and variables, including extensions, see the [GNU SETL Library Reference](#).

SETL Variations

*** To be a summary of how GNU SETL differs from CIMS SETL and from SETL2. May point to www.settheory.com and Jack's unfinished book, perhaps commenting on the different approach embodied in his Tk chapter *vs.* how I use wish pumps.

*** This might also be the place for some brief mention of SETL's relatives such as Python, or at least a few words to mention that my thesis goes into that a bit.

Links

Other GNU SETL Manuals

For how to compile and run programs, see the [GNU SETL User Guide](#).

For a tutorial introduction to the SETL language, see the [GNU SETL Tutorial](#) [stub].

For a reference treatment of the language supported by GNU SETL, see the [GNU SETL Language Reference](#) [stub].

For full details on the built-in operators and functions in GNU SETL, see the [GNU SETL Library Reference](#). (Not a stub. It is currently the main guide to GNU SETL.)

For commentary on the internal interfaces and implementation of GNU SETL, see the [GNU SETL Implementation Notes](#) [stub].

For how to extend the language with datatypes and intrinsic functions derived from C headers and libraries, see the [GNU SETL Extension Guide](#) [stub].

If you are a developer helping to maintain GNU SETL, please see the [GNU SETL Maintenance Manual](#) [stub].

External Links

Finn Wilcox used to have a [SETL Wiki](#) with example programs and other useful resources. [An Invitation to SETL](#) is an enthusiastic endorsement of SETL contributed by Roberto De Leo to the *Linux Journal* on December 28, 2004.

Robert Dewar's quite comprehensive and very readable little book, [The SETL Programming Language](#), is still a pretty accurate guide to the core SETL language supported by GNU SETL if you leave out the stuff about macros (GNU SETL uses an adaptation of the GNU C Preprocessor instead), backtracking, the data representation sublanguage, and old-style modules (GNU SETL adopts SETL2 packages).

GNU SETL supports both SETL and SETL2 syntax, so the examples in the adaptation of Robert Dewar's book by Robert Hummel, [A Gentle Introduction to the SETL2 Programming Language](#), should work without modification. Note that SETL2 classes and closures are not supported by GNU SETL.

The classic textbook on SETL, *Programming with Sets: An Introduction to SETL*, by Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., and Schonberg, E. (Springer-Verlag, New York, 1986), has been partly rewritten for SETL2 at <https://www.settheory.com> as *Programming in SETL*. The version of SETL described in the 1986 textbook is referred to in the GNU SETL documentation as *CIMS SETL*.

[SETL2](#), a close relative of SETL created by Kirk Snyder and developed further by Toto Paxia, is described in [The SETL2 Programming Language](#) and [SETL2: An Update on Current Developments](#) (both 1990).

[SETL-S](#), by Robert Dewar and Julius VandeKopple, is a high-performance SETL subset implementation for DOS systems.

PSETL, by Zhiqing Liu, uses the GNU SETL translator to feed an interpreter that records intermediate program states using a space-efficient encoding scheme based on the *persistent* data structures of Driscoll, Sarnak, Sleator, and Tarjan in *J.Comp.Sys.Sci.* **38**, 1989. This

allows execution histories to be reviewed in complete detail, including *all* values ever acquired by *all* variables. Its graphical interface is also a good pedagogical and debugging tool.

[Robin Pourtaud](#) has a very nice web site for [ISETL and ISETLW](#), a descendant of Gary Levin's *Interactive SETL* and a Windows-specific version thereof. Under Ed Dubinsky's influence, ISETL has been widely used in the teaching of discrete mathematics.

The [Slim](#) language by Herman Venter is another interesting cousin of SETL.

Work on set-theoretic languages and programming continues on various fronts, and [Enrico Pontelli](#) at the New Mexico State University maintains a site [Programming with {Sets}](#) containing an extensive bibliography, information on workshops and conferences, links to implementation sites, etc. [Gianfranco Rossi](#) at the Università di Parma likes to use sets for Constraint Logic Programming as shown at his [JSetL Home Page](#).

The [dB thesis](#), *SETL for Internet Data Processing* (2000), a sort of SETL Rationale, describes most of the extensions to SETL that have shaped GNU SETL, and shows how to use them in practice. It cites a great deal of other past work too.

Administrivia

Copying GNU SETL

GNU SETL's source is licensed under the Free Software Foundation's GNU Public License (GPL). See the file `COPYING` in the top-level directory of the GNU SETL source distribution for the rules on copying and modifying GNU SETL.

If you distribute GNU SETL executables (`setl`, `setlcpp`, `setltran`) or documentation, e.g. by posting the files on an FTP or Web site, please concurrently provide similar access to the GNU SETL source distribution from which those files were built.

All the source code for the GNU SETL system should be available under <https://setl.org/setl/>; please contact dB <David.Bacon@nyu.edu> otherwise.

Reporting Bugs

If you find a bug in GNU SETL, please send email to dB <David.Bacon@nyu.edu>, including the output of `setl --version`, your command-line arguments, the environment variables, the input if possible, the output you got from the `setl`, `setlcpp`, or `setltran` command, and some description of the output you expected. For details on those commands, see the *GNU SETL User Guide*.

Index

A

Ada/Ed..... 1
arbitrary *vs.* random..... 6

B

bugs, reporting..... 11

C

CIMS SETL..... 1
COPYING..... 11

D

De Leo, Roberto..... 9
Dewar, R.B.K..... 1, 9
distributing GNU SETL source..... 11
Dubinsky, E..... 9, 10

F

Free Software Foundataion..... 11

G

GNU Public License (GPL)..... 11

H

high-level language..... 1
Hummel, Robert..... 9

I

ISETL, ISETLW..... 10
iterator..... 4

J

JSetL..... 10

L

Levin, Gary..... 10
Linux Journal..... 9
Liu, Zhiqing..... 9

M

modifying GNU SETL..... 11
multi-map..... 1

P

Paxia, Toto..... 9
Pontelli, Enrico..... 10
Pourtaud, Robin..... 10
PSETL..... 9

Q

quantified expressions..... 5

R

readability..... 1
Rossi, Gianfranco..... 10

S

Schonberg, E..... 9
Schwartz, J.T..... 1, 2, 9
setl command..... 7
SETL Wiki..... 9
SETL-S..... 9
SETL2..... 9
Slim..... 10
Snyder, Kirk..... 9
source code, GNU SETL..... 11

V

value semantics..... 4
VandeKopple, Julius J..... 9
Venter, Herman..... 10

W

Wilcox, Finn..... 9