

GNU SETL Om

On the World's Most Wonderful Programming Language, with Extensions
Edition 0.1.2, for GNU SETL Version 0.1.2
Updated 5 June 2009

by dB

Table of Contents

1	Introduction	1
1.1	General characteristics of SETL	1
1.2	History	1
1.3	GNU SETL.....	3
2	To the Student of Programming Languages ..	4
3	SETL Variations	7
4	Links	8
4.1	Other GNU SETL Manuals	8
4.2	External Links	8
5	Administrivia	10
5.1	Copying GNU SETL.....	10
5.2	Reporting Bugs	10
	Index	11

1 Introduction

1.1 General characteristics of SETL

SETL is a general-purpose programming language whose distinctive feature is that finite sets, and in particular maps, are fundamental to its semantics and syntax. For example, the set-valued expression

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

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 . (The first vertical bar here is best read as *such that*, and the second as a pause.)

Tuples, like sets, have first-class syntactic status in SETL. For example, 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 therefore no language-defined 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 $f("age")$ 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 might 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` and `while`. The iterators found in loop headers also form the basis for the predicates and the set and tuple formers (“comprehensions”) that give SETL many of the advantages of a strictly functional language.

For more on iterators, maps, and the “strict value semantics” of SETL, see [Chapter 2 \[To the Student of Programming Languages\]](#), page 4.

1.2 History

SETL began as a tool for the high-level expression of complex algorithms. It quickly 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.

However, as Robert Dewar likes to point out, the readability of a program is much more important than the speed with which it can be written, and this is as true for production code as it is for a prototype. SETL, happily, is based on notations that mathematicians have found helpful in communicating with each other, as this quotation from Jack Schwartz’s original monograph, *Set Theory as a Language for Program Specification and Programming*, illustrates:

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.

To this day, sets and maps abound in high-level presentations of algorithms. SETL helps programmers approach the ideal of “presentation-grade” programs in their daily work by making such programs directly executable.

In its focus on clarity of expression, SETL also helps programmers avoid getting mired in premature optimization. Quoting Schwartz again, this time from *On Programming*:

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 much more comprehensive survey of SETL history, including numerous bibliographic references, please see [dB's Ph.D. thesis](#).

1.3 GNU SETL

GNU SETL is an implementation of SETL. The goal has always been to have a `setl` command (see the *GNU SETL User Guide*) that would play well in a POSIX environment, allowing it to be used as casually and conveniently in a shell script as one might use `awk`, `sed`, or `grep`. One might now add `python` to that list. Even with an external interface consisting of little more than basic I/O, `setl` did indeed prove to be a valuable tool in a seemingly endless variety of roles, from combinatorial experiments to routine scientific data processing. However, as described in the abovementioned thesis, there came a time when it seemed worthwhile to add new facilities for working more directly with processes, signals, timers, and sockets, allowing whole systems to be built up easily as hierarchies of small programs in client-server and parent-child relationships.

So that is where GNU SETL stands today. It presents as a cooperative little command for processing what may best be regarded as a particularly elegant and powerful scripting language.

2 To the Student of Programming Languages

SETL is conducive to precise, concise, readable programming largely by virtue of its fierce resistance to the addition of machine-oriented clutter.

One consequence of this attitude is that there are no pointers in the language. It is natural for those who have never lived without them to wonder how such a language could be convenient, or even possible, to write significant programs in.

The answer is the maps. Even where you might be tempted to think pointers are necessary, such as in a linked data structure, it is almost always better to make the nodes the range elements of a map over logical keys. The keys (domain elements) will in general represent what you are interested in much more directly and naturally than a set of artificial pointers can. Moreover, you are spared the hazards of aliases—no two variables can ever refer to the same object.

This aspect of SETL is sometimes called its *strict value semantics*, and has some useful consequences. For example, assignment of a set or tuple works by “deep” copying, which makes transmission of entire data structures between programs, even over the network, a trivial matter. Another helpful consequence is that there is no harm in updating a structure over which you are iterating, because you are really only iterating over a copy.

Maps are central to SETL syntax. Here is a trivial example, in which word frequencies are tallied:

```
count := {};          -- empty mapping
for word in split (getfile stdin) loop
  count[word] += 1;  -- init to 1 or tally
end loop;
```

Here, the `count` map is built up by incrementing `count[word]` for each `word` in the tuple produced by `split` (whose second argument defaults to “whitespace”). If `count[word]` is undefined (om in SETL terms) when we go to add an integer to it, it is taken as 0, so this is a complete program although it doesn’t actually do anything with the map it builds.

We could make a histogram from the `count` map:

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

This shows one way of iterating over a map. Here, successive ordered pairs are broken out from `count` and assigned “in parallel” to `[word, freq]`, meaning `word` gets the string and `freq` gets the associated integer, on each iteration.

EXERCISE 1. For map `m`, what does

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

represent?

You probably see immediately that it is the inverse of `m`, but is that colon a misprinted vertical bar?

Actually, it is not. The general form of the inside of a *set former* such as this (or a *tuple former* which employs curly braces instead of square brackets) is:

expression : *iterator* | *condition*

(There can even be multiple *iterators*, separated by commas (for nested iteration (inner-most to the right)).)

The *iterator* loops to assign successive values to some variables (like *word*, or *x* and *y*). Each time around, the *condition* is evaluated in terms of those variables. If it turns out to be *true*, the *expression* is evaluated and the result is added to the set or tuple being constructed.

This general form 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 loops and set and tuple formers, are the predicates. These are boolean-valued expressions consisting of *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. Note that 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 the idiomatic

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

for swapping the contents of variables *a* and *b*.

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

To beginners in SETL, especially those with a strong background in formal set theory, the deliberate resemblance between SETL iterators and the similar parts of a comprehension or mathematical predicate can lead to confusion. It is important to remember that an iterator is *executed* as a loop over some set, tuple, or string value, and assigns values to iteration variable(s) as it goes. The bound variables in a mathematical predicate or comprehension, by contrast, are just tags to be used in statements characterizing the members of a set—often an infinite and/or implicit set (“universe”) at that.

Moreover, the common iterator form

x in *s*

is simply a (boolean-valued) membership test in contexts requiring an expression rather than an iterator.

EXERCISE 2. Given sets *s*₁ and *s*₂, 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 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 sophisticated compiler 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` 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 fairness in the selection, and SETL has a separate operator for 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 this 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 the iterator and a boolean expression: “`freq = count(word)`” is certainly true within the body of the above loop.

Incidentally, this function-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, respectively. This is also consistent with the notation for element access (“subscripting”) for these types.

For more details on these and other iterator forms such as

```
ys = f{x}
```

(note the braces around `x`), the reader is referred to the Tutorial and the Language Reference. See [Chapter 4 \[Links\]](#), page 8.

3 SETL Variations

*** *To be written.*

*** To be a summary of how GNU SETL differs from CIMS SETL and from SETL2. May point to www.settheory.com and Jack's book in progress, perhaps commenting on the different approach embodied in his Tk chapter *vs.* how I just use wish pumps. Collate with whatever goes in `index.html`, moving most of what's there here.

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

4 Links

4.1 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, see the *GNU SETL Language Reference* [stub].

For detailed semantic information on the built-in operators and functions, see the *GNU SETL Library Reference*.

For commentary on the internal interfaces and implementation, 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].

4.2 External Links

Finn Wilcox's *SETL Wiki* is a great SETL resource centre.

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 and is at <http://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*.

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.

ISETLW, by John Kirchmeyer, is a descendant of Gary Levin's “Interactive SETL”, *ISETL*. Under Ed Dubinsky's influence, these SETL variants have 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 Ph.D. thesis](#), *SETL and the Internet*, 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.

5 Administrivia

5.1 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.

*** This is still (31 May 2009) the plan, not quite yet the fact:

All the source code for the GNU SETL system should be available under <http://setl.org>; dB <bacon@cs.nyu.edu> would very much like to hear about it if you ever find that not to be the case.

5.2 Reporting Bugs

If you find a bug in GNU SETL, please send email to dB <bacon@cs.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 10

C

comprehension (set, tuple) 5
 ‘COPYING’ 10

D

Dewar, R.B.K. 1, 8
 distributing GNU SETL source 10
 Dubinsky, E. 8

F

Free Software Foundataion 10

G

GNU Public License (GPL) 10

H

high-level language 1

I

ISETL, ISETLW 8
 iterator 4

J

JSetL 8

K

Kirchmeyer, John 8

L

Levin, Gary 8
 Liu, Zhiqing 8

M

modifying GNU SETL 10
 multi-map 1

P

Paxia, Toto 8
 Pontelli, Enrico 8
 POSIX 3
 predicate 5
 PSETL 8

R

readability 1, 4
 Rossi, Gianfranco 8

S

Schonberg, E. 8
 Schwartz, J.T. 1, 2, 8
`set1` command 3
 SETL Wiki 8
 SETL-S 8
 SETL2 8
 Slim 8
 Snyder, Kirk 8
 source code, GNU SETL 10

V

value semantics 1, 4
 VandeKopple, Julius 8
 Venter, Herman 8

W

Wilcox, Finn 8