The Denotational Semantics of SETL

Michael N. Condict
Courant Inst., New York University
June 9, 1983

1. Introduction

We give here a denotational definition of a large subset of the SETL programming language as defined in [1]. It is not intended to be an introduction to SETL for the casual reader -- it is much too concise and has too little redundancy of expression to serve that purpose. It should instead be used as a precise supplement to [1], to resolve the ambiguities, inconsistencies and gaps that are bound to occur in a large, nonmathematical description. The definition is given in the language of standard set theory augmented by typed λ-calculus (and other) notation. As is usual with such definitions, induction on the context-free syntax of programs is used to define a meaning function, which maps SETL program fragments to the functions they denote. It should be noted that some liberties have been taken with the syntax rules as given in [1], although the language is the same. Specifically, replacement of right-hand-side nonterminals by their definitions was used where convenient, leaving the grammar smaller (but less abstract).

2. Mathematical Primitives

Before getting started with the definition it is helpful to make it quite clear what portion of mathematics is being taken for granted. This is done not only to clarify the distinction between the meta-language (mathematics) and the object-language (SETL) but to anticipate those detractors who feel that the best definition of a programming language is one that consists of a translation to an abstract machine and a description of how to execute that machine. The belief seems to be that mathematics is such a large and complicated language that no clarity is gained by giving a definition that amounts to no more than an interpreter for the language, but which happens to be written in mathematics instead of some simpler "programming language".

Well, it would be foolish (and useless) to deny that a denotational definition is anything more than an interpreter program, but it is not at all clear that mathematics as a programming language is as complicated as many of the machines and formal systems that have been introduced as semantic tools, at least not the small portion of mathematics employed in most denotational definitions. And it *is* clear that (classical) mathematics is more widely understood, and has a larger body of facts known about it than most of these newer machines, which are often defined by resorting to mathematics anyway.

The mathematical language used in this definition is ordinary Zermelo-Franckel set theory, augmented with notational "abbreviations" (not all of which are shorter than the expressions they denote). For a very short, but complete, definition of this language, the reader is referred to an introductory logic text such as [2], giving a set of proof rules for first-order logic together with the axioms of ZF set theory (including the axiom of choice, which we need). We mention such a proof system not because it is a practical way to learn the language of set theory, but to anticipate and counter the argument that mathematics is not precisely defined or "executable." To execute a set theoretic expression, thus discovering the value it denotes, one need only implement a theorem prover and then supply it with the axioms of set theory and an axiom describing the expression of interest. Of course, such a method is not guaranteed to terminate for all expressions, but this is true of any interpreter executing programs in a sufficiently powerful language.

Although the only truly primitive operations of set theory are those of first-order logic, together with ∈ (set membership), we will use many more than these without defining them. It is a straightforward (and entertaining) task to give definitions of all such operations in terms of the truly primitive ones, but we do not bother to do so here.

All that we will give is a list of the mathematical notations, operations and sets that are used without definition. The only deviation from the standard meaning of these terms is that all the primitive sets are assumed, for convenience, to be mutually disjoint, so that, for instance, one can tell the difference between the natural number 3, the integer 3 and the rational 3.0. One can assume, if desired, that all the elements of these sets are paired with a "type-symbol", allowing such distinctions to be made. In the same manner, we later assume that the sets and n-tuples formed from these basic sets are all distinct from one another; that is, no integer is a set, an n-tuple is distinguishable from a set, and so on. Again, assume that type-symbols are used to keep track of such distinctions.

Sets:

$$\begin{aligned} Bool&=\{false,true\}\\ \omega&=\{0,1,2,\cdots\}\\ Int&=\{\cdots,-2,-1,0,1,2,\cdots\}\\ Real&=\text{(a subset of the rational numbers)}\end{aligned}$$

Set Ops:	$e\in S$, $S\subseteq T$, $\{e_1,\ldots,e_k\}$, $\{e[x] \mid x\in S \text{ and } b[x]\}$, $S\cup T$, $S\cap T$, $S-T$, $Power(S)$, $\lvert S\rvert$.
Logic:	α and β, α or β, $\neg\alpha$, α implies β, $\exists x{:}\alpha[x]$, $\forall x{:}\alpha[x]$, α iff β, $x=y$.
Functions:	$f(e)$, $f(e_1,\ldots,e_k)$, $(\lambda x\in S.e[x])$, $S\rightarrow T$, $Domain\ f$, $Range\ f$,

$$f(x)=\begin{cases} e_1[x], \text{ if } b_1[x],\\ \cdots\\ e_k[x], \text{ if } b_k[x].\end{cases}$$

Numeric:	$x+y$, $x-y$, $x\cdot y$, x/y, $\min(x,y)$, $\max(x,y)$, $\lvert x\rvert$, $\sin x$, $\cos x$, $\tan x$, $\exp x$, $\cdots$
Sequences:	(n-tuples)
	$<e_1,\ldots,e_k>$, $s\cdot t$ (concatenation), s_i (selection), S^* (repetition).

Note: Throughout the definition, function application is often written without parentheses, except for grouping, so, i.e., fx denotes f applied to x. The precedence of function application is taken to be higher than any other mathematical operator in the definition, so that $fx+gy$ denotes $(fx)+(gy)$, not $f(x+(gy))$. Also, $(f\ x_1\ x_2\ \cdots\ x_k)$ should be parenthesized as $(\cdots((f\ x_1)\ x_2)\ \cdots\ x_k)$, and can usually be viewed as the (Curried) application of f to arguments $x_1,\ldots,x_k$ rather than as the application of a k^{th} order functional.

Other notation used above that might need explanation includes $Power(S)$, which is usually written with a script P and denotes the set of all subsets of S, and $\lambda x\in S.e[x]$ (where S is a set and $e[x]$ represents an expression possibly containing references to x), which denotes the function that, given any x in S, produces $e[x]$. Every occurrence of x within $e[x]$ that is not within an inner $\lambda x\cdots$ expression, or other expression that binds x, is a reference to the argument of the function, and is said to be a *bound* variable occurrence (in the same manner that x is bound in the formula $\forall x{:}\phi$).

From now on, when we are making a definition, we will write:

$$new\ notation =_{df} old\ notation$$

where *new notation* is the notation or variable being given meaning, and *old notation* is its definition. We use $=_{df}$ to avoid confusion with the equality symbol =, which will often appear as part of the definition. Even though the new notation may be shown surrounded by parentheses () for clarity, the parentheses will sometimes be omitted in later usages of the notation, where ambiguity would not result.

When we are defining a notation that gives a binding to one or more variables, as is the case with the λ notation, above, we will emphasize this by writing $e[x]$ (or $e[x_1,...,x_k]$) instead of just e for each constituent expression of the new notation within which x is bound. The bracketed x is part of the meta-language in which the definition is being made and will not appear in subsequent uses of the notation. The only real semantics associated with the use of $[x]$ is that if we introduce an expression as $e[x]$, then $e[t]$ means "the expression e with every free occurrence of x replaced by the expression t, in the usual manner."

3. Domain Definitions

Now that we have laid the meta-language on the table, we can begin to define new sets and other useful objects. We will start by giving, in this section, the domains that will be most important in the SETL definition. These are the sets whose elements represent SETL data objects and the meaning of SETL constructs.

To be more specific, *Obj* represents the set of values that SETL variables can take on; *St* is the set of possible states during the execution of a SETL program and is nothing more than a mapping from addresses (locations in *Loc*) to SETL values (elements of *Obj*); *Cont* is the set of continuations, functions from states to states that are used soley to define **goto** 's. A continuation is passed as an argument to the meaning of some SETL constructs, representing the state transition that needs to be done after the execution of the construct in order to finish executing the program in which the construct is embedded (see the definition of labels and **goto**).

The special location *val* does not refer to any explictly named SETL object, but is used to hold the value resulting from executing expressions. That is, rather than specifying the meaning of an expression as both a function from states to states and as a function from states to a value, we need only give the former. The value of the expression is taken from the *val* field of the state produced by executing the expression. One can think of the *val* location as an accumulator (or R0, for those familiar with certain machine languages). The other special locations (ones that are not SETL identifiers) are used for various purposes documented later.

[3.1] $CharOrd =_{df} <c_1, \ldots, c_n>$
where each c_i is a symbol called a "*character*"

[3.2] $Char =_{df} \{c \mid \exists i : CharOrd_i = c\}$

[3.3] $Str =_{df} Char^*$

[3.4] $Atom =_{df}$ (a countable set disjoint from Str join Bool join Int join Real)

[3.5] $A \rightarrow_f B =_{df} \{f \in A \rightarrow B \mid \text{Domain } f \text{ is finite}\}$

[3.6] $Power_f\, S =_{df} \{x \subseteq S \mid x \text{ finite}\}$

[3.7] $D_0 =_{df} Bool \cup Int \cup Real \cup Str \cup Atom$

[3.8] $D_{i+1} =_{df} D_i \cup (\omega \rightarrow_f D_i) \cup Power_f\, D_i$

[3.9] $Obj =_{df} \{om\} \cup \bigcup_{i \in \omega} D_i$

[3.10] $Loc =_{df} \{val, yield\_cont, return\_cont\} \cup$ (the set of non-reserved SETL identifiers)

[3.11] $St =_{df} Loc \rightarrow Obj$

[3.12] $Cont =_{df} St \rightarrow St$

The special object *om*, above, which represents the SETL value **om** , is some value that is disjoint from all of the D_i. It is also used to indicate error conditions when rules of SETL are violated.

It may surprise some readers to see that the set of locations, *Loc*, is defined as the set of SETL identifiers (variable names). This can be done because SETL is a pure value language, with no pointers or call-by-address. Hence, there is no need for both an environment, mapping variable names to locations, and a store, mapping locations to values. Instead we use just a mapping from variable names (which *are* locations, for us) to values, and call each such mapping a *state*. In the interest of abstractness we keep the rest of this document ignorant of the fact that the locations are SETL identifiers, except where the information is necessary in the manipulation of declarations.

In the definition of *Obj*, above, D_0 represents the set of SETL "scalar" objects, D_1, the scalars, sets of scalars, and tuples of scalars and, in general, D_i contains all SETL objects whose nesting depth (of sets/tuples within sets/tuples) is between 0 and i. Note that, while a SETL set is represented as a mathematical set, it would not be appropriate to use n-tuples as the representation of SETL tuples, which can have positions vacant in the beginning or middle of the sequence. Instead they are represented more naturally as finite functions whose domain is a subset of ω, the natural numbers. So, if T is such a finite function, $T(i)$ represents the i^{th} element of the SETL tuple represented by T.

Now, all that remains is to define the highest level domain, the one that contains the denotations of SETL program fragments. This is the set *Mng*, but before we can define it we have to talk about the handling of functions. In SETL, functions share the same name scope with global variables (there are no nested functions). It is thus appropriate to have a domain that maps identifiers to *either* function denotations (to be defined) *or* value denotations (members of *Obj*). This would automatically enforce the restriction that a function may not have the same name as a global variable. Such a domain could also map label identifiers, the third category of named SETL entity, to continuations (members of *Cont*), indicating the state transformation to be performed upon branching to that label. We define this domain, *Env*, below and call it the set of "environments."

What are the denotations of functions? We have arranged to have them be the same as denotations of any other SETL program fragments, by the following scheme. We assume that, when a function is called, the *val* field of the environment contains the tuple of its actual arguments and that, when it returns, the *val* field contains the result value. In this way, there is nothing more to a function than a mapping from states to states, which explains the definition of *Func*, the set of function denotations.

Now we can define *Mng* as a function from environments to environments in the absence of *goto*'s, and as a function from continuation/environment pairs to environments otherwise.

[3.13] $Func =_{\text{df}} St \rightarrow St$

[3.14] $Env =_{\text{df}} Loc \rightarrow (Func \cup Cont \cup Obj)$

[3.15] $Mng =_{\text{df}} (Env \rightarrow Env) \cup (Cont \rightarrow (Env \rightarrow Env))$

The portion of an environment that maps identifers to continuations or functions is completely static in SETL. That is, there is no whole SETL construct (at least according to the way we have broken SETL into constructs) the execution of which can change this environment. This is because SETL does not allow functions to be treated as objects, even to the extent of being allowed as parameters, and does not have label variables. The result of this staticness is that we can get away with representing the meaning of a SETL function as a mapping from states to states ($St \rightarrow St$) rather than a mapping from environments to environments ($Env \rightarrow Env$). To convert it from the former to the latter we simply extend it with the appropriate portion of the identity function on environments. So, if we have a particular environment e within which we wish to execute a function whose meaning is $f \in St \rightarrow St$ we first augment f to obtain the following $f' \in Env \rightarrow Env$. Let $e \mid Obj$ denote the largest restriction of e that has a range that is a subset of Obj (that is, the portion of the environment e that is a state) and define

$$f'(e) =_{\text{df}} \lambda x \in Loc. \begin{cases} f\ (e \mid Obj)(x), & \text{if } e(x) \in Obj, \\ e(x), & \text{otherwise.} \end{cases}$$

Now we can apply f' to e to produce the environment e' that can differ from e only on those locations that were mapped to objects by e. In the section where we describe function application (and the *goto* statement) we introduce a concise notation for this conversion.

4. Preliminary Definitions and Abbreviations

Before any of the syntactic abbreviations we introduce a very important semantic one. In order to make the meta-language look more natural, to say nothing of conciseness, it is crucial to avoid repetitive references to a environment variable s (recall that SETL programs are viewed as functions from environments to environments) when it is obvious which environment we are talking about and all that is being done is to apply it to some location x. For $x \in Loc$ and $s \in Env$ we write x instead of $s(x)$ when possible subject to the following rule of interpretation: wherever an expression e with value in Loc is used where only an expression with value in Obj makes sense (is in the domain of the function to which e is an argument), replace it by $s(e)$ where s is the parameter of the innermost surrounding lambda expression of the form $\lambda s \in Env.g(e,s)$. However, this replacement is to be done only after all other abbreviations have been expanded, so that all lambda variables have been made explicit. In order to allow us to give a meaningful example of this rule let us first introduce the following useful definition. For any $f \in A \rightarrow B$, $x \in A$, $e \in B$, let

$$[4.1]\ f[x/e] =_{\mathrm{df}} \text{the function } f' \text{ s.t.} f'(y) = \begin{cases} e, & \text{if } x=y, \\ f\ y, & \text{o.w.} \end{cases}$$

Now we can write, say, the function that adds 1 to the integer at location x as:

$$Increment =_{\mathrm{df}} \lambda s \in Env.s[x/x+1]$$

which by the previous rule, is an abbreviation for

$$\lambda s \in Env.s[x/(s\ x)+1]$$

Most of the rest of the abbreviations will be introduced where they are first used, so that their motivation is clear, but the following are important enough to deserve special mention (Though they are shown surrounded by parentheses, these abbreviations will usually be used with parentheses omitted. Also, the syntax will sometimes be informal and allowed to spill over into the surrounding text -- the meaning, we trust, will be clear).

The first of these have to do with λ notation. Although for reasons of simplicity we prefer to allow only unary functions as a primitive notion, we obtain the effect of having multiple argument functions by an abbreviation called Currying:

$$\lambda x_1 \in S_1, \ldots, x_k \in S_k.e[x_1, \ldots, e_k] =_{\mathrm{df}} \lambda x_1 \in S_1.(\lambda x_2, \ldots, x_k \in S_k.e[x_1, \ldots, e_k])$$

for $k \geq 2$. As an example of this mechanism, if we want a function that takes two arguments, one from A and the other from B, and produces something in C we represent it as

$$\lambda x \in A.\lambda y \in B.e[x,y]$$

which is denoted by

$$\lambda x \in A, y \in B.e[x,y]$$

and is a function of type $A \rightarrow (B \rightarrow C)$.

Now some abbreviations familiar to applicative programmers:

$$[4.2]\ (\text{if } b \text{ then } e_1 \text{ else } e_2) =_{\mathrm{df}} \begin{cases} e_1, & \text{if } b \text{ is a true condition,} \\ e_2, & \text{o.w.} \end{cases}$$

$$[4.3]\ (\text{let } x = e_1 \text{ in } e_2[x]) =_{\mathrm{df}} (\lambda x \in \{e_1\}.e_2[x])(e_1)$$

$$[4.4]\ (e_2[x], \text{ where } x = e_1) =_{\mathrm{df}} \text{let } x = e_1 \text{ in } e_2[x]$$

$$[4.5]\ (\text{letrec } f = (\lambda x \in S.e_1[f,x]) \text{ in } e_2[f]) =_{\mathrm{df}} \text{let } f = \text{l.f.p.}(\lambda f \in S \rightarrow T.\lambda x \in S.e_1[f,x]) \text{ in } e_2[f]$$

$$[4.6]\ (\text{for } x \in S, \text{ letrec } f(x) = e_1[f,x] \text{ in } e_2[f]) =_{\mathrm{df}} \text{letrec } f = (\lambda x \in S.e_1[f,x]) \text{ in } e_2[f]$$

where T, the range of f above, contains $\{e[f,x] \mid x \in S\}$. The least fixed point (l.f.p.) of a functional $H \in ((S \rightarrow T) \rightarrow (S \rightarrow T))$ is defined as follows:

$f_0 =_{\mathrm{df}} H(\perp)$, where $\perp$ is $\{\}$, the totally undefined function.
$f_{i+1} =_{\mathrm{df}} H(f_i)$,
$l.f.p.(H) =_{\mathrm{df}} \bigcup_{i \in \omega} f_i$

For those not familiar with the notion of least fixed point of recursive functionals and how they relate to recursive function definitions see, e.g., [3]. Readers who wish to take our word for it may simply be informed that

$$l.f.p.(\lambda f \in S \rightarrow T.\lambda x \in S.e[f,x])$$

is the function that we understand informally to be defined by:

$$f(x) =_{\mathrm{df}} e[f,x]$$

although the former has meaning even when the recursive use of f appears to be circular, in that its argument is not always smaller (or simpler by some linear measure) than x itself. The point of using least fixed points is just that property: they allow us to define the meaning of, say, a **while** loop as a recursive function which "calls" itself endlessly precisely in those cases where the **while** loop does not terminate.

The previous definitions were all of a general enough nature that they can be said to have nothing to do with the semantics of SETL -- they are general-purpose tools. It is now time to give away a little more detail about the "internals" of the SETL definition. Each SETL statement or expression is mapped by the definition to a function $f \in Mng$, which is either a function from environments to environments or a function which given a continuation and a environment produces a environment. The idea is that simple constructs, ones that cannot execute jumps out of themselves, will be mapped to $(Env \rightarrow Env)$, while ones that can cause the disruption of normal sequential execution will be mapped to $(Cont \rightarrow (Env \rightarrow Env))$ and passed a continuation consisting of the portion of the program following the statement. This is so that they can choose to ignore the continuation passed to them when they wish to execute a jump. The point of saying all this is that it is time to show how such functions should be composed to get sequential execution, say, when two statements follow each other in a statement list. The notation used, as is the case with much of the other notation, will mimic standard procedural programming language syntax. Thus, we will use $f_1; f_2$ to mean the composition (in the manner to be specified) of the meaning function f_1 with the meaning function f_2. The functions f_1, f_2 will usually, but not always, have been derived as the meaning of SETL program fragments. (Note: In the definition of ; , below, it was convenient to extend what is really a binary operator to arbitrary, possibly empty, lists of of meaning functions.) Let $k \geq 0$, with $f_1, \ldots, f_k \in Mng$ and define:

$$[4.7]\ (f_1; \cdots ; f_k) =_{\mathrm{df}} \begin{cases} (\lambda s \in Env.s), & \text{if } k=0, \\ f_1, & \text{if } k=1, \\ f_1 \circ f_2, & \text{if } k=2, \\ f_1; (f_2; \cdots ; f_k), & \text{if } k \geq 3 \end{cases}$$

where, for $f,\ g \in Mng$,

$$f \circ g =_{\mathrm{df}} \begin{cases} \lambda s \in Env.(g\ (f\ s)), & \text{if } f,\ g \in Env \rightarrow Env, \\ \lambda c \in Cont, s \in Env.(g\ c\ (f\ s)) & \text{if } f \in Env \rightarrow Env,\ g \in Cont \rightarrow (Env \rightarrow Env), \\ \lambda c \in Cont, s \in Env.(f\ (c \circ g)\ s), & \text{if } f \in Cont \rightarrow (Env \rightarrow Env),\ g \in Env \rightarrow Env, \\ \lambda c \in Cont, s \in Env.(f\ (g\ c)\ s), & \text{if } f,\ g \in Cont \rightarrow (Env \rightarrow Env) \end{cases}$$

Notice that we are defining the meaning of $f \circ g$ in a way such that it has its standard meaning of function composition when f and g are both in $Env \rightarrow Env$, but has a more complicated definition when continuations are involved.

Since SETL is a procedural language, assignment (state change) is clearly an important notion. We have decided to use a familiar notation for it. For $x \in Loc$, $v \in Obj$,

$$[4.8]\ (x := v) =_{\mathrm{df}} \lambda s \in Env.s[x/v]$$

Note that x must be a location and v an expression in Obj. Hence, by the rule for abbreviating $s(x)$ to x,

$$\begin{aligned} (x := x+1) &= (\lambda s \in Env.s[x/x+1]) \\ &= (\lambda s \in Env.s[x/(s\ x)+1]) \end{aligned}$$

It will often be convenient to construct a function that, given a environment, applies one of several different functions to it, depending on the value of that environment, or even depending on the value of previous environments. This could be written as, say:

$$\lambda s \in Env.F[s,f,g]$$

where $F[s,f,g]$ is an expression that tests s and then either applies f to it or applies g to it (see, e.g., the semantics of the **if** statement). However it is better to make this more concise by using what could be called labeled environments:

$$[4.9]\ (s{:}f[s]) =_{\mathrm{df}} \begin{cases} \lambda s \in Env.(f[s]\ s), & \text{if } f \in Env \rightarrow Env, \\ \lambda c \in Cont, s \in Env.(f[s]\ c\ s), & \text{if } f \in (Cont \rightarrow (Env \rightarrow Env)) \end{cases}$$

The above notation is useful only when f is defined with s as a free variable, otherwise, $(s{:}f)=f$. Much obfuscating parenthesization can be avoided by integrating this notation with the ; notation:

$$[4.10]\ (f_1; \cdots ; f_k; s{:}g_1; \cdots ; g_n) =_{\mathrm{df}} f_1; \cdots ; f_k; (s{:}g_1; \cdots ; g_n)$$

Another notion that occurs repeatedly in the definition is that of executing several SETL program fragments in unspecified order, and then using all the results. For example, in the expression $e_1 + e_2$ the two sub-expressions e_1, e_2 are evaluated this way, then the two results are added together. The following definition provides a convenient way to specify this notion. It composes a list of meaning functions into one that, in addition to executing all the component functions, leaves in the special location *val*, the sequence of values produced by each of the functions (see explanation of *val*, above). For $k \geq 2$, with $f_1, \ldots, f_k \in Mng$,

```
[4.11]
do $<> =_{\mathrm{df}} (val := <>)$,
do $<f_1> =_{\mathrm{df}} (f_1;\ val := <val>)$,
do $<f_1, \ldots, f_k> =$
    if $\forall$ permutations $<i_1, \ldots, i_k>$ of $<1, \ldots, k>$,
        $(f_{i_1}; \cdots ; f_{i_k}) = (f_1; \cdots ; f_k)$
    then
            $f_1;\ s_1{:}\ f_2;\ \cdots\ s_{k-1}{:}f_k;$
        $s_k{:}\ val := <s_1\ val, \ldots, s_k\ val>$
    else $val := om$
```

We will need, in various places, the notion of picking an arbitrary member of a set (specifically, in the definition of the **arb** function). In SETL this notion is completely nondeterministic -- not only is the member chosen not a pure function of the set, but it is not even necessarily repeatable from one execution of the program to the next. We use an oracle function *arb* that *is* a function of the set to model this behavior, for now, but it is not general enough.

$$[4.12]\ \text{arbitrary } S =_{\mathrm{df}} arb(S)$$

In order to allow SETL tuples to be defined in a natural looking manner, while remaining consistent with the desired SETL semantics, we alter the meaning of λ terms that are denotations of SETL tuples as follows. For any finite $S \subseteq \omega - \{0\}$,

$$[4.13]\ \lambda x \in S.e[x] =_{\mathrm{df}} f, \text{ where } f(y) = \begin{cases} e[y], \text{ if } y \in S, \\ om, \text{ otherwise.} \end{cases}$$

Now we invoke the right reserved earlier to keep all the primitive sets and the sets and tuples formed from them disjoint, and define some type testing operations. For any v, v_1, $v_2 \in Obj \cup Func$,

$[4.14]$ v is a set $=_{\mathrm{df}} \exists i \in \omega{:} v \in Power_f(D_i)$

$[4.15]$ v is a tuple $=_{\mathrm{df}} \exists i \in \omega{:} v \in \omega \rightarrow_f D_i$

$[4.16]$ v is a string $=_{\mathrm{df}} v \in Char^*$

$[4.17]$ v is a pair $=_{\mathrm{df}}$ v is a tuple and $Domain\ v = \{1,2\}$

$[4.18]$ v is a map $=_{\mathrm{df}}$ v is a set and $\forall x \in v{:}(x$ is a pair$)$

[4.19] v is a single-valued map $=_{df}$ v is a map and $\forall x,y\in v{:}x(1)=y(1)$ implies $x=y$

[4.20] v is a function $=_{df}$ $v\in Func$

Finally, a few useful functions dealing with SETL objects. For $v,v_1,v_2\in Obj$ and $c\in Char$:

[4.21] $\mathrm{Pair}(v_1,v_2) =_{df} \lambda x\in\{1,2\}.\text{if } x=1 \text{ then } v_1 \text{ else } v_2$

[4.22] null tuple $=_{df}$ $\lambda x\in\{\}.om$

[4.23] tuple length of v $=_{df}$ if v is a tuple then max ($Domain$ v) else om

[4.24] $Ord(c) =_{df}$ the i such that $c=CharOrd_i$

5. The Meaning Function

The meaning of most SETL constructs will be defined to be a function from environments to environments ($Env\rightarrow Env$), namely the function that describes the environment that results from executing the fragment in any initial environment. The meaning of program construct u will be written $[\![u]\!]$, so it is always the case that $[\![u]\!]\in Mng$. Occurrences of variables within u are meta-variables that denote arbitrary SETL variables, or arbitrary members of other classes of SETL constructs, as noted.

Now we finally get to the definition of the SETL meaning function. The definition is made by induction on the syntax of SETL constructs. Consequently, we can claim that the domain of the function defines also the syntax of SETL programs, as long as we understand the vocabulary of tokens, and we do not need to give a separate syntactic definition. (For convenient reference, however, the Appendix contains a summary of the syntax of SETL.) This differs from most other denotational definitions, which are usually defined on the abstract syntax of a language, requiring a separate definition of the concrete syntax.

Each case in the inductive definition is labeled by the name of the SETL construct the meaning of which is being defined. The simplest constructs are presented first, that is, the definition proceeds from the lowest level nonterminals in the SETL grammar to the nonterminal "SETL_program" at the highest level (we do not deal with modules and libraries).

5.1. primary

The SETL primaries are the bottom level expressions, at least with respect to operator precedence. They include the numeric and string literals, array subscripts and function calls (both having the same syntax), and the set and tuple formers.

5.1.1. literal (int_tok, real_tok, string)

For any int_tok n, let $nval$ be the integer represented by n in the standard manner. Then

[5.1] $[\![n]\!] =_{df} (val{:}=nval)$

For any real_tok r, let $rval$ = the real number represented by r in the standard manner. Then

[5.2] $[\![r]\!] =_{df} (val{:}=rval)$

For any string $'c_1 \cdots c_k'$, where $c_i\in Char$,

[5.3] $[\![' c_1 \cdots c_k ']\!] =_{df} (val{:}=<c_1,\ldots,c_k>)$

5.1.2. selection

For a primary (not a lhs) consisting of a variable or function identifer x, and selectors $s_1,\ldots,s_k$, $k\geq 0$,

```
[5.4] [[x s_1 ··· s_k]] =_df
            do <[[es_1]], . . . ,[[es_k]]>;
            s: get ts_1 ··· ts_k of s x
```

where

$$es_i=\begin{cases} e, & \text{if } s_i=(e) \text{ or } s_i=\{e\}, \\ [e_1,\ldots,e_n], & \text{if } s_i=(e_1,\ldots,e_n) \end{cases},$$

$$ts_i=\begin{cases} (), & \text{if } s_i=(e) \text{ or } s_i=(e_1,\ldots,e_n), \\ \{\}, & \text{if } s_i=\{e\} \end{cases},$$

The expression es_i is simply the expression inside the brackets of the selector s_i, while ts_i is the pair of brackets. We also take care here of the rule that says a list of expressions in a selector is the same as putting a tuple as the selector (transparently to the user, we allow this rule to apply to SETL functions as well as SETL tuples and maps, to obtain the simplicity that all SETL functions may be treated as though they were unary).

It remains to define the notation: get $ts_1 \cdots ts_k$ of $s\ x$, which denotes a certain function from environments to environments. Its purpose (when $k>0$) is to access the value denoted by a sequence of selector expressions applied to the value $s\ x$. When it is applied to a environment s, there had better be a tuple of at least k values in $s\ val$, because it will use these values as indices or arguments. In the usage of get $\cdots$, above, the tuple of values was put in the val field by the "do $\cdots$" function.

First, the base case; when there are no selectors we already have the value, unless it is a function, in which case we call it. For $v\in Obj\cup Func$,

[5.5] get of $v =_{df}$ s:if v is a function then call funct $v(<>)$
else $val:=x$

For $k\geq 1, v\in Obj\cup Func$,

get $ts_1 \cdots ts_k$ of $v =_{df}$
s: get $ts_1 \cdots ts_{k-1}$ of v;
s':$val:=$ get1 ts_k, $(s\ val)_k$ of $(s'\ val)$

where

$$\text{get1 } (),\ i \text{ of } v =_{df} \begin{cases} val:=v\ i, & \text{if } v \text{ is a tuple,} \\ val:=\text{the unique } y \text{ s.t. Pair}(i,y)\in v, & \text{if } v \text{ is a map,} \\ \text{call funct } v(i), & \text{if } v \text{ is a function.} \end{cases}$$

$$\text{get1 } \{\},\ i \text{ of } v =_{df} \begin{cases} val:=om, & \text{if } v \text{ is a tuple,} \\ val:=\{y \mid pair(i,y)\in v\}, & \text{if } v \text{ is a map,} \\ val:=om, & \text{if } v \text{ is a function.} \end{cases}$$

As can be seen, we have defined get recursively. To get a value from v using k selectors, we get a value from v using the first $k-1$ selectors, then we use get1 to apply the k^{th} selector to the result.

In order to complete the definition of get1, we have to give the most important part of its definition (or at least, the most interesting), namely "call funct". For $v\in St\rightarrow St$ and $i\in Obj$

[5.6] call funct $v(i) =_{df}$ $val:=i$; $(v \mid St/Env)$

where $(v \mid St/Env)$ converts a function on states to a function on environments as follows. For any $v\in St\rightarrow St$ and $s\in Env$,

[5.7] $s \mid Obj =_{df}$ the largest restriction of s that has its range in Obj

[5.8] $(v \mid St/Env)(s) =_{df} \lambda x\in Loc.\begin{cases} v\ (s \mid Obj)(x), & \text{if } s(x)\in Obj, \\ s(x), & \text{otherwise.} \end{cases}$

Informally, the difference between v and $(v \mid St/Env)$ is just that the latter, when given an environment that is not a state, applies v to the "state" portion of the environment and leaves the rest of the environment (the portion mapping locations to non-objects) unchanged.

5.1.3. iterated set, tuple

For any expressions e_1, e_2, e_3,

```
[5.9] $[\![\{e_1..e_2\}]\!] =_{df}$
          do $<[\![e_1]\!],[\![e_2]\!]>$;
          $val := \{x \in Int \mid min(val_1,val_2) \le x \le max(val_1,val_2)\}$

[5.10] $[\![\{e_1,e_2..e_3\}]\!] =_{df}$
          do $<[\![e_1]\!],[\![e_2]\!],[\![e_3]\!]>$;
          $val := \{x \in Int \mid \exists n \in Int$ s.t.
                    $x = val_1 + n*(val_2 - val_1)$ and
                    $min(val_1,val_3) \le x \le max(val_1,val_3)\}$

[5.11] $[\![[e_1..e_2]]\!] =_{df}$
          do $<[\![e_1]\!],[\![e_2]\!]>$;
          $val := (\lambda i \in \{1, \ldots, |val_2 - val_1| + 1\}.$
                    $val_1 + (i-1)*sgn(val_2 - val_1))$,

[5.12] $[\![[e_1,e_2..e_3]]\!] =_{df}$
          do $<[\![e_1]\!],[\![e_2]\!],[\![e_3]\!]>$;
          $s$: (let $incr = val_2 - val_1$,
                $n =$ if $incr = 0$ then 0 else
                        $1 + |floor((val_3 - val_1)/incr)|$
             in
                $val := (\lambda i \in \{1, \ldots, n\}.val_1 + (i-1)*incr))$,
```

$$\text{where } sgn(n) = \begin{cases} 1, & \text{if } n > 0 \\ 0, & \text{if } n = 0 \\ -1, & \text{if } n < 0 \end{cases}$$

For any expression e and iterator i,

```
[5.13] $[\![[e:i]]\!] =_{df}$ iterate over $i$ producing $[\![e]\!]$;

[5.14] $[\![\{e:i\}]\!] =_{df}$
          iterate over i producing $[\![e]\!]$;
          $val := Range\ val$                    (Convert the tuple produced to a set)
```

where, for any left-hand sides $x, y, x_1, \cdots$, expressions $e, e_1, \cdots$, simple iterators $si, si_1, \cdots$, Boolean expression b and $f \in Mng$, "iterate over..." is defined as:

```
[5.15] iterate over $x$ in $e \mid b$ producing $f =_{df}$
    $[\![e]\!]$;
    $s$:if $s\ val$ is a set then $val :=$ a tuple containing the elts of $val$
       else $skip$;
    $s{:}val := <s\ val, 1,$ null tuple$, 1>$;
    letrec $loop = s_0{:}x := (s_0\ val)_1\ (s_0\ val)_2$;
                 $s$:if $s\ x \ne om$ then $[\![b]\!]$ else $val := false$;
                 $s$:if $s\ val$ and $s\ x \ne om$ then
                       $f$;
                       $s{:}val := <(s_0\ val)_1, (s_0\ val)_2,$
                              $(s_0\ val)_3 \cup \{<(s_0\ val)_4, s\ val>\}, (s_0\ val)_4 + 1>$;
                    else $skip$
```

$s{:}val{:=}\langle(s\ val)_1,(s\ val)_2+1,(s\ val)_3,(s\ val)_4\rangle$;
s:if $(s\ val)_2 \leq$ tuple length of $(s\ val)_1$ then *loop* else *skip*
in *loop*

[work here -- discuss above def.]

[5.16] iterate over $y{=}e(x)$ | b producing $f =_{\mathrm{df}}$
s: iterate over $[x,y]$ in $[\![e]\!]$ | b producing f

[5.17] iterate over $y{=}e\{x\}$ | b producing $f =_{\mathrm{df}}$
[work here]

[5.18] iterate over $si_1, \ldots, si_k$ | b producing $f =_{\mathrm{df}}$
iterate over si_1 producing
iterate over $si_2, \ldots, si_k$ | b producing f;
$val{:=}$let $k{=}(length\ of\ val)$ in $val_1 \cdot \cdots \cdot val_k$

[5.19] iterate over $si_1, \ldots, si_k$ producing $f =_{\mathrm{df}}$
iterate over $si_1, \ldots, si_k$ | **true** producing f

5.1.4. enumerated set, tuple

For any expressions $e_1, \ldots, e_k$, $k \geq 0$,

[5.20] $[\![\{e_1, \ldots, e_k\}]\!] =_{\mathrm{df}}$
do $\langle[\![e_1]\!], \ldots, [\![e_k]\!]\rangle$;
$val{:=}\{val_1, \ldots, val_k\}$

[5.21] $[\![[e_1, \ldots, e_k]]\!] =_{\mathrm{df}}$
do $\langle[\![e_1]\!], \ldots, [\![e_k]\!]\rangle$;
$val{:=}(\lambda i{\in}\{j{\in}\{1, \ldots, k\} \mid val_j{\neq}om\}.val_i)$

5.1.5. from_expr

For any two variable identifiers x and y,

[5.22] $[\![x$ **from** $y]\!] =_{\mathrm{df}} [\![y$ **less** $:=(x{:=}$**arb** $y)]\!]$; $val{:=}x$

[5.23] $[\![x$ **fromb** $y]\!] =_{\mathrm{df}}$ s:let $i{=}\min(Domain\ (s\ y)){\in}$
$y{:=}y-\{\langle i,z\rangle \mid \langle i,z\rangle{\in}s\ y\}$;
$x{:=}s\ y\ (i)$;
$val{:=}x$;

[5.24] $[\![x$ **frome** $y]\!] =_{\mathrm{df}} [\![x{:=}y(\#y)]\!]$; $s{:}[\![y(\#y){:=}$ **om** $]\!]$; $val{:=}s\ val$

For any two lhs's $x\ xs_1, \ldots, xs_k$ and $y\ ys_1, \ldots, ys_n$,

[5.25] $[\![x\ xs_1, \ldots, xs_k$ **from** $y\ ys_1, \ldots, ys_n]\!] =_{\mathrm{df}}$???

[5.26] $[\![x\ xs_1, \ldots, xs_k$ **fromb** $y\ ys_1, \ldots, ys_n]\!] =_{\mathrm{df}}$???

[5.27] $[\![x\ xs_1, \ldots, xs_k$ **frome** $y\ ys_1, \ldots, ys_n]\!] =_{\mathrm{df}}$???
[work here]

5.1.6. case_expr

For expressions $e, v_1, \ldots, v_k$ and constant lists $cl_1, \ldots, cl_k$,

```
[5.28]
⟦case $e$ of
    ($cl_1$): $v_1$, . . . ,($cl_k$): $v_k$
    else $v_{k+1}$
 end ⟧ $=_{df}$

              ⟦expr case $e$ of
                    ($cl_1$):yield $v_1$;  · · · ($cl_k$):yield $v_k$;
                    else yield $v_{k+1}$;
                 end case ;
              end ⟧
```

For Boolean expression lists $el_1, \ldots, el_k$ and expressions $v_1, \ldots, v_k$,

```
[5.29]
⟦case  of
    ($el_1$): $v_1$, . . . ,($el_k$): $v_k$
    else $v_{k+1}$
 end ⟧ $=_{df}$
          ⟦expr case  of
                 ($el_1'$):yield $v_1$;  · · · ($el_k'$):yield $v_k$;
                 else yield $v_{k+1}$;
              end case ;
          end ⟧
```

where, for any Boolean expression list $el=e_1, \ldots, e_n$,
$el'=e_1$ **or** $\cdots$ **or** e_n

5.1.7. if_expr

For expressions b, e_1 and e_2,

[5.30] ⟦**if** b **then** e_1 **else** e_2 **end** ⟧ $=_{df}$ ⟦**expr if** b **then yield** e_1; **else yield** e_2; **end if** ; **end** ⟧

5.1.8. prog expr

For statements $s_1, \ldots, s_k$,

[5.31] ⟦**expr** $s_1 \cdots s_k$**end** ⟧ $=_{df}$
$\lambda c \in Cont, s \in Env.$(with temp env $<yield\_cont \rightarrow c>$ do ⟦$s_1 \cdots s_k$⟧) c s
(see *EnvironmentManipulation*, below for def. of with-temp-env)

where *yield_cont*, like *val*, is a special field of the environment that is not the address of any explicitly named SETL object. It indicates what action should be done after a **yield** statement. We execute the statement list of the **expr** in a modified environment that maps *yield_cont* to the continuation of the **expr** itself.

5.1.9. sys_vals

The first three of these are just predefined constants, while the last, **newat**, is a nullary function that returns a value that is not only guaranteed to be different from any SETL object obtainable any other way, but is also different from the value returned by any previous call to **newat**. It is analogous to the LISP gensym function.

[5.32] ⟦**true**⟧ $=_{df}$ $val{:=}true$

[5.33] ⟦**false**⟧ $=_{df}$ $val{:=}false$

[5.34] ⟦ **om** ⟧ $=_{df}$ $val{:=}om$

[5.35] ⟦**newat**⟧ $=_{df}$ $val{:=}\text{arbitrary}\, x \in Atom;\ used\_ats{:=}used\_ats \cup \{x\}$

5.2. term

The terms of SETL are either primaries or unary operations, such as **arb** x, **sin** x and so on, including the (one argument) reductions of binary operators $(+/e)$. We have already defined the meaning of primaries, so we need only take care of the last two here.

In order to give a concise semantic definition for the unary operators we first define the mathematical function corresponding to each SETL unary operator. For those operators that are overloaded, we give the correspondence for each type of operand, using the convention that x,y,z are operands of any type, m,n are numeric operands (real or integer), $c,c_1,\cdots$ are characters, s,t are character strings, A,B are sets, S,T are tuples and α,β are Boolean operands.

Another implicit (but important) assumption is that all of these functions are extended to arbitrary $x \in Obj$ outside their normal domain, and they return *om* when supplied with such an "illegal" x.

SETL Operator	Corresponding Mathematical Function
$\#T$	if T is a tuple then tuple length of T else *om*
$+n$	n (identity)
$-n$	$-n$ (negation)
abs n	$\lvert n \rvert$
abs s	if $\lvert s \rvert = 1$ then $Ord(s_1)$ else *om*
acos n	acos n
arb A	arbitrary 'A' (see def. of roman arbitrary , above)
asin n	sin n
atan n	atan n
ceil n	$\lceil n \rceil$
char n	if $n \le \lvert CharOrd \rvert$ then $CharOrd_n$ else *om*
cos n	cos n
domain A	if A is a map then $\{x \mid Pair(x,y) \in A\}$ else *om*
even n	n is an even number
exp n	e^n (the mathematical constant e)
fix n	if $n \in Real$ then the $n' \in Int$ s.t. $\lfloor n \rfloor = n'$ else *om*
float n	if $n \in Int$ then the $n' \in Real$ s.t. $n = n'$ else *om*
floor n	$\lfloor ? \rfloor$
is_atom x	if $x \in Atom$ then *true* else *false*
is_boolean x	if $x \in Bool$ then *true* else *false*
is_integer x	if $x \in Int$ then *true* else *false*
is_map x	if x is a map then *true* else *false*
is_real x	if $x \in Real$ then *true* else *false*
is_set x	if x is a set then *true* else *false*
is_string x	if x is a string then *true* else *false*
is_tuple x	if x is a tuple then *true* else *false*
log n	log n
not α	$\neg\alpha$ (logical negation)
odd n	n is an odd number
pow A	$Power(A)$
random A	?
range A	if A is a map then $\{y \mid Pair(x,y) \in A\}$ else *om*
sign n	sign n
sin n	sin n
sqrt n	?
str x	any string of characters s s.t. $[\![s]\!] = (val := x)$
tan n	tan n
tanh n	tanh n
type x	if $x \in Atom$ then "*ATOM*" else
	if $x \in Bool$ then "*BOOLEAN*" else

```
if $x \in Int$ then "INTEGER" else
if $x \in Real$ then "REAL" else
if $x$ is a set then "SET" else
if $x$ is a string then "STRING" else
if $x$ is a tuple then "TUPLE"
```

Now we can refer to the above chart in giving the semantics of expressions that contain unary operators. In the following definitions, let d, e be terms, op be a unary operator and f_{op} be the function corresponding to op (with the operand type taken into account at each use of f). Then, for any expression e,

[5.36] $[\![op\ e]\!] =_{df} [\![e]\!];\ val := f_{op}(val)$

```
[5.37] $[\![op / e]\!] =_{df}$
    $[\![e]\!]$;
    $s$:if $\neg val$ is a tuple then $val := om$
        else $val :=$(letrec $reduce = \lambda n \in \omega$.
                    if $n = 0$ then $v$ else
                        $f_{op}(reduce(n-1),\ s\ val(n))$
                in $reduce$(tuple length of $val$),
                where $v =$if $s\ val\ (1) \in Int \cup Real$ then 0
                    else if $s\ val\ (1)$ is a set then $\{\}$
                    else if $s\ val\ (1)$ is a tuple then null tuple
                    else if $s\ val\ (1)$ is a string then $<>$)
```

5.3. expression

Expressions consist either of a binary operator between two terms $(d + e)$, the assigning form of a binary operator $(x + := e)$, the two argument form of reduction $(d + / e)$, or a quantifier (exists $x \mid b$).

As with the unary operators we first define the mathematical function corresponding to each SETL binary operator, again using the convention that x,y,z are operands of any type, m,n are numeric operands (real or integer), $c, c_1, \cdots$ are characters, s,t are character strings, A,B are sets, S,T are tuples and α,β are Boolean operands.

SETL Operator	Corresponding Mathematical Function
$m+n$	$m+n$ (addition)
$s+t$	$s \cdot t$ (concatenation)
$S+T$	$\lambda i \in \{1..$(tuple length of S + tuple length of T)$\}$. if $i \leq$ tuple length of S then $S(i)$ else $T(i-$ tuple length of S)
$A+B$	$A \cup B$
$m-n$	$m-n$ (numeric subtraction)
$A-B$	$A-B$ (set subtraction)
$m*n$	$m \cdot n$ (multiplication)
$n*s$	s^n (n repetitions of s)
$s*n$	(same as $n*s$)
$n*T$	if $n<0$ then om else if $n=0$ then null tuple else (same as $T+(n-1)*T$)
$T*n$	(same as $n*T$)
m/n	m/n (division)
$m**n$	m^n (power)
$m<n$	$m<n$ (numeric comparison)
$s<t$	$\exists i \in \omega: Ord(s_i) < Ord(t_i)$ (lexicographic comparison)

$m \leq n$	$m \leq n$ (numeric comparison)
$s \leq t$	$s = t$ or $\exists i \in \omega : Ord(s_i) < Ord(t_i)$ (lexicographic comparison)
$x = y$	$x = y$ (equality)
$m \geq n$	$m \geq n$ (numeric comparison)
$s \geq t$	$s = t$ or $\exists i \in \omega : Ord(s_i) > Ord(t_i)$ (lexicographic comparison)
$m > n$	$m > n$ (numeric comparison)
$s > t$	$\exists i \in \omega : Ord(s_i) > Ord(t_i)$ (lexicographic comparison)
$x /= y$	$x /= y$ (inequality)
$x ? y$	($if\, x \neq om$ then x else y)
α **and** β	α and β (logical disjunction)
m **atan2** n	if $n \neq 0$ then arctan m/n else sign $m \cdot \pi/2$
m **div** n	$\lfloor m/n \rfloor$
α **impl** β	α implies β
A **in** B	$A \in B$
c **in** s	$\exists i \in \omega : c = s_i$
x **in** T	$\exists i \in \omega : x = T(i)$
A **incs** B	$A \subseteq B$
A **less** x	$A - \{x\}$
A **lessf** B	$A - \{Pair(y,z) \mid y \in B\}$
m **max** n	$\max(m,n)$
m **min** n	$\min(m,n)$
m **mod** n	m mod n
x **notin** y	(same meaning as **not** x **in** y)
n **npow** A	$\{x \subseteq A \mid \lvert x \rvert = n\}$
A **npow** n	(same meaning as n **npow** A)
α **or** β	α or β (logical conjunction)
A **subset** B	$A \subseteq B$
A **with** x	$A \cup \{x\}$

In the following definitions, let d, e be terms, op be a binary operator and f_{op} be the function corresponding to op (with the operand type taken into account at each use of f). Then

[5.38] $[\![d\ op\ e]\!] =_{df}$ do $<d,e>$; $val := f_{op}(val_1, val_2)$

For any variable or function identifier x and expression e,

[5.39] $[\![x\ op{:=}\ e]\!] =_{df} [\![e]\!]$; $x := f_{op}(x, val)$

For any variable or function identifier x, expression e and selectors $s_1, \ldots, s_k$, $k \geq 1$,

```
[5.40] $[\![x\ s_1, \ldots, s_k\ op{:=}\ e]\!] =_{df}$
           do $<[\![es_1]\!], \ldots, [\![es_k]\!], [\![e]\!]>$;
           $s$: get $ts_1 \cdots ts_k$ of $s\ x$;
           $val := f_{op}(val, s\ val_{k+1})$;
           set $ts_1 \cdots ts_k$ of $x$,
```

```
[5.41] $[\![d\ op/\ e]\!] =_{df}$
            do $<[\![d]\!], [\![e]\!]>$;
            $s$:if $\neg val_2$ is a tuple then $val := om$
             else $val :=$(letrec $reduce = \lambda n \in \omega$.
                                  if $n = 0$ then $s\ val_1$ else
                                     $f_{op}(reduce(n-1), s\ val_2(n))$
                        in $reduce$(tuple length of $val_2$))
```

5.3.1. assignment_expr

For variable identifier x and expression e,

[5.42] $[\![x := e]\!] = [\![e]\!]$; $x := val$

For variable identifier x, selectors $s_1, \ldots, s_k$ and expression e,

```
[5.43] $[\![x\ s_1 \cdots s_k{:=}e]\!]=$
            do $<[\![es_1]\!], \ldots, [\![es_k]\!], [\![e]\!]>$;
            set $ts_1 \cdots ts_k$ of $x$,
    where
       set $ts_1 \cdots ts_k$ of $x =_{\mathrm{df}}$
            $s$: get $ts_1 \cdots ts_{k-1}$ of $x$;
            set1 $ts_k$, $(s\ val)_k$ of $val$ to $(s\ val)_{k+1})$
            $val{:=}<(s\ val)_1, \ldots, (s\ val)_{k-1}, val>$
            set $ts_1 \cdots ts_{k-1}$ of $x$;
```

where

$$\text{set1 (), } i \text{ of } v \text{ to } e =_{\mathrm{df}} \begin{cases} val{:=}v[i/e], & \text{if } v \text{ is a tuple and } e\neq om, \\ val{:=}v-\{<i,x>\mid i\in Domain\ v\}, & \text{if } v \text{ is a tuple and } e=om, \\ val{:=}v-\{Pair(i,x)\mid Pair(i,x)\in v\}\cup\{Pair(i,d)\}y & \text{is a single-valued map and } e\neq om \\ val{:=}v-\{Pair(i,x)\mid Pair(i,x)\in v\}, & \text{if } v \text{ is a single-valued map and } om \end{cases}$$

$$\text{set1 \{\}, } i \text{ of } v \text{ to } e =_{\mathrm{df}} \begin{cases} val{:=}om, & \text{if } v \text{ is a tuple,} \\ val{:=}???, & \text{if } v \text{ is a map.} \end{cases}$$

[work here -- finish definition of set1]

It is probably necessary to talk a little about the definition of "set" and "set1", above. Informally, the meaning of "set" $ts_1 \cdots ts_k$ of $x =_{\mathrm{df}}$ is "set the portion of x selected by the first k components of the *val* tuple to the $k+1$th component of the *val* tuple. That is, "set" expects the *val* field to contain a $k+1$-tuple of values, the first k being selector values, and the last being the value that is to be assigned. It fetches a structured value by using the first $k-1$ selectors on x. Then it modifies an element of this value using the kth selector. Finally it sets the portion of x just fetched to the modified value. For left-hand sides $x_1, \ldots, x_k$ and expression e,

[5.44] $[\![[x_1, \ldots, x_k]{:=}e]\!] =_{\mathrm{df}}$
[To be supplied]

5.4. stmt body

5.4.1. assignment, from_expr, proc_call

(These constructs are also expressions and have the same meaning as their expression forms. See *primary*, above, for the definitions.)

5.4.2. goto_statement

Here is where continuations become meaningful. To execute a *goto*, we look up the continuation associated with the label identifier, map this continuation into an environment-transition function f with $(\ldots\mid St/Env)$ and produce the function that, given a continuation and an environment, ignores the continuation and applies f to the environment. For label identifier l,

[5.45] $[\![\textbf{goto } l]\!] =_{\mathrm{df}} \lambda c\in Cont, s\in Env.((s\ l)\mid St/Env)\ s$

5.4.3. yield_statement

In executing the surrounding prog_expr, we mapped the special location *yield_cont* to the continuation of the prog_expr, so we just obtain this function, and use it instead of the continuation of the **yield** statement. For expression e,

```
[5.46] $[\![\textbf{yield } e]\!] =_{\mathrm{df}} \lambda c\in Cont, s\in Env.([\![e]\!];$
                                   $((s\ yield\_cont)\mid St/Env))\ s$
```

5.4.4. if_statement

For Boolean expression b and statement lists sl_1, sl_2,

```
[5.47] [[if b then sl₁ else sl₂ end  ··· ; ]] =df
            [[b]];
            s: if (s val)=true then [[sl₁]] else [[sl₂]]
```

5.4.5. case_statement

5.4.6. loop_statement

We will describe all the loops except the **for** loop in terms of a complete loop statement. First, we define the meaning of a loop with some of its optional clauses left out. Let *ls* be a (non **for**) loop statement with, possibly, some optional clauses, such as **doing** *stmts*, missing. Let *ls'* be the complete loop statement obtained from *ls* by filling in the missing clauses with the corresponding clauses in this list (most of which have a reserved word followed by a null statement list):

```
init
doing
while true
step
until false
term
```

Then we define:

$[\![ls]\!] =_{df} [\![ls']\!]$

It remains to define the semantics of a complete loop statement. Let $sl_1, \ldots, sl_k$ be statement lists and b_1, b_2 be Boolean expressions.

```
[5.48]
[[loop  init sl₁
     doing sl₂
     while b₁
     step sl₃
     until b₂
     term sl₄
  do
     sl₅
 end loop ]] =df

     [[sl₁]];
     letrec l=[[sl₂]]; [[b₁]];
        if val then
           [[sl₃]]; [[sl₅]][[b₂]];
           if val then l else skip
         else skip
      in l
     [[sl₄]];
```

Now we define the **loop for** statement. Let *sl* be a statement list and *i* be an iterator.

```
[5.49] [[loop for i do sl end loop ]] =df
            iterate over i producing [[sl]];
```

5.5. Environment Manipulation

Before going on with the semantic definition, we will introduce and discuss some abbreviations involving manipulation of the environment. They will be useful both in the definition of lists of labeled statments (see *smts* below) where we need to introduce labels into the environment (mapped to continuations) and in the definition of a Body, where we need to introduce variables, functions and constants into the environment.

We will use two abbreviations, "with temp env" and "with rec temp env', which are analogous to let and letrec, respectively. The first is fairly simple. For $v_1, \ldots, v_k \in Loc$, $f \in Mng$ and $e_1, \ldots, e_k \in Loc \cup Func \cup Cont$,

```
[5.50] with temp env $v_1 \rightarrow e_1, \ldots, v_k \rightarrow e_k$ do $f =_{df}$
                $s{:}v_1{:}=e_1;\ \cdots;\ v_k{:}=e_k;$
                $f;$
                $v_1{:}=(s\ v_1);\ \cdots;\ v_k{:}=(s\ v_k)$
```

It just assigns several values to the locations v_i, executes the function f then restores the original values of the v_i.

Now we will need an extension of the $f[x/e]$ notation to allow the modification of f at more than one point. For partial functions $f, g \in A \rightarrow B$,

[5.51] update f with $g =_{df}$ $\lambda x \in A.$if $x \in Domain\ g$ then $g\ x$ else $f\ x$

So $f[x/e]=$ update f with $\{<x,e>\}$.

Now the hard part. Informally, we want
[work here -- more discussion]
For $v_1, \ldots, v_k \in Loc$ and $e, e_1, \ldots, e_k \in Mng$, $k \geq 1$,

```
[5.52]
with rec temp env
    $v_1 \rightarrow e_1, \ldots, v_k \rightarrow e_k$
do $e =_{df}$
        $s'$: letrec $f_1 = \lambda s \in St.$((with temp env $v_1 \rightarrow f_1, \ldots, v_k \rightarrow f_k$
                                do $e_1$)
                                (update $s'$ using $s$)$\mid Obj$)
                ...
                $f_k = \lambda s \in St.$((with temp env $v_1 \rightarrow f_1, \ldots, v_k \rightarrow f_k$
                                do $e_k$)
                                (update $s'$ using $s$)$\mid Obj$)
            in with temp env $v_1 \rightarrow f_1, \ldots, v_k \rightarrow f_k$
              do $e$
```

[More discussion of above sorely needed.]

5.6. stmts (statement lists)

The semantics of labels and, hence, **goto** are partially defined here as well. We have limited the scope of a label to the statement it labels, together with all the statements in the list of statements immediately containing the labeled statement. Thus the reach of a **goto** statement *gs* is limited to statements containing *gs* (directly or indirectly) and statements that are side-by-side in a statement list with a statement containing *gs*. Another way to say this is that transfer of control may only go, first, from more deeply nested to surrounding statements, followed optionally by a jump from one statement in a statement list to another.

For $k \geq 1$, let $sl_0, \ldots, sl_k$ be lists of semicolon-terminated statement bodies (unlabeled statements), and $l_1, \ldots, l_k$ be identifiers.

```
[5.53] $[\![sl_0\ \ l_1{:}sl_1\ \ \cdots\ \ l_k{:}sl_k]\!] =_{df}$
        with rec temp env $l_1 = [\![sl_1\ \ sl_2\ \ \cdots\ \ sl_k]\!]$,
                          $l_2 = [\![sl_2\ \ sl_3\ \ \cdots\ \ sl_k]\!]$,
```

$$\begin{array}{l} \quad\quad\quad\quad \cdots \\ \quad\quad\quad\quad l_k = [\![sl_k]\!] \\ \text{do } [\![sl_0 \;\; sl_1 \;\; \cdots \;\; sl_k]\!] \end{array}$$

where, for statement bodies $s_1, \ldots, s_n$,

$$[\![s_1; \;\cdots; \; s_n]\!] =_{\text{df}} [\![s_1]\!]; \;\cdots\; ; [\![s_n]\!]$$

5.7. program/procedure body

Here we give the core of the environment manipulation in SETL. There are only two levels of scope in SETL (at least in our subset), consisting of the global variables that remain allocated throughout the execution of the program, and local variables that are allocated and released in conjunction with function calls. Both cases are handled with the same definition here. Note that for convenience we assume all constants are declared before all variables and that no variables are used without being declared. We could, of course, describe syntactic transformations that would map any program/procedure body not in this form to one that is, observing the rule that undeclared variables are taken to be local, not global (unless they occur in the body of the main program).

In this section it is convenient to introduce a construct that is not in the SETL grammar as such, although it is a piece of a SETL program. Let "refined statement list" refer to a list of statements followed by zero or more refinements:

$$Refined\_stmt\_list \rightarrow Stmts \; Identifier{::}Stmts \; \cdots$$

Let $c_1, \ldots, c_k, v_1, \ldots, v_n$ be identifiers, $ce_1, \ldots, ce_k$ be constants and sl be a refined statement list. Then we define

$$\begin{array}{l} [5.54] \\ [\![\textbf{const } c_1 = ce_1, \ldots, c_k = ce_k; \\ \;\textbf{var } v_1, \ldots, v_n; \\ \;sl]\!] =_{\text{df}} \\ \quad\quad\quad \text{do } < [\![ce_1]\!], \ldots, [\![ce_k]\!] >; \\ \quad\quad\quad \text{with temp env } c_1 \rightarrow val_1, \ldots, c_k \rightarrow val_k, v_1 \rightarrow om, \ldots, v_n \rightarrow om \\ \quad\quad\quad \text{do } [\![sl]\!] \end{array}$$

To handle the case where initial values are specified using the **init** clause, we simple transform such clauses into assignment statements that are performed before the statement list is executed. Let $c_1, \ldots, c_k, v_1, \ldots, v_n$ be identifiers, $ce_1, \ldots, ce_k$ be constants, $ve_1, \ldots, ve_r$ be constants and sl be a refined statement list. Also, let $\{v_{i_1}, \ldots, v_{i_r}\} \subseteq \{v_1, \ldots, v_n\}$ and define

$$\begin{array}{l} [5.55] \\ [\![\textbf{const } c_1 = ce_1, \ldots, c_k = ce_k; \\ \;\textbf{var } v_1, \ldots, v_n; \\ \;\textbf{init } v_{i_1} := ve_1, \ldots, v_{i_r} := ve_r; \\ \;sl]\!] =_{\text{df}} \\ \quad\quad\quad [\![\textbf{const } c_1 = ce_1, \ldots, c_k = ce_k; \\ \quad\quad\quad\; \textbf{var } v_1, \ldots, v_n; \\ \quad\quad\quad\; v_{i_1} := ve_1; \;\cdots; \; v_{i_r} := ve_r; \\ \quad\quad\quad\; sl]\!] \end{array}$$

The next section gives the meaning of a refined statement list, completing the semantic definition of program/proc bodies.

5.8. refined stmt list

Refinements in SETL are nothing but identifiers that abbreviate statement lists. Their meaning could be defined syntactically, as if they were macros, but with the semantic weapons we already have deployed, it is just as easy to define them semantically, as parameterless procedures. For identifiers $r_1, \ldots, r_k$ and statement lists $sl_0, \ldots, sl_k$, $k \geq 1$:

```
[5.56] [[$sl_0$  $r_1$::$sl_1$ · · · $r_k$::$sl_k$]] $=_{df}$
            with rec temp env $r_1 \rightarrow [\![sl_1]\!]$
                                      . . .
                                   $r_k \rightarrow [\![sl_k]\!]$
            do $[\![sl_0]\!]$
```

Our definition is meaningful even when refinements refer recursively to themselves, whereas the macro definition would not be (and SETL does not allow such references, perhaps for this reason).

5.9. SETL program

A SETL program consists of some global declarations and global statements (b_p, below) followed by function declarations. The strategy we use to define its meaning is straightforward. We define the meaning of a program with n function declarations in terms of a program with $n-1$. The semantics of a program with no function declarations is then defined trivially from the semantics of its body.

For any identifiers $p, f_1, \ldots, f_n$ and bodies $b_p, b_1, \ldots, b_n$, $n \geq 1$,

```
[5.57]
[[program $p$;
     $b_p$
   proc $f_1(x_1, \ldots, x_k)$;
     $b_1$
   end proc $f_1$;
   . . .
   proc $f_n(z_1, \ldots, z_p)$;
     $b_n$
   end proc $f_n$;
 end program $p$;]] $=_{df}$
                           with rec temp env
                             $f_1 \rightarrow$(with temp env $x_1 \rightarrow val_1, \ldots, x_k \rightarrow val_k$
                                 do $[\![b_1]\!]$)
                             . . .
                             $f_n \rightarrow$(with temp env $z_1 \rightarrow val_1, \ldots, z_p \rightarrow val_p$
                                 do $[\![b_n]\!]$)
                           do [[program $p$;
                                   $b_p$
                                end program $p$;]]
```

Finally, the base case; when there are no function declarations,

```
[5.58]
[[program $p$;$b_p$end program $p$;]] $=_{df}$ $[\![b_p]\!]$
```

6. Summary

The language described above is not just a subset of SETL, but were it not that the syntax rules do not allow it, the definition gives meaning to many useful program fragments that are in an extended SETL -- one that is much closer to being an expression language. For instance, note that the definition of the **for** loop is such that if a **for** loop were allowed as an expression, it would have a value consisting of the tuple of values produced by each iteration. These values, in turn are well defined, even though the body of a **for** loop is a list of statements, not an expression.

[More to be supplied. End of document, for now.]

7. References

(1) Dewar, R. K., Schonberg, E. & Schwartz, J. T., *Higher Level Programming*, Computer Science Dept., Courant Inst., New York, 1981.

(2) Bell, J. & Machover M., *A Course in Mathematical Logic*, North-Holland, Amsterdam, 1977.

(3) Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.