SETL for Internet Data Processing

by

David Bacon

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Computer Science

New York University

January, 2000

Jacob T. Schwartz (Dissertation Advisor)

For my children

Acknowledgments

First of all, I would like to thank my advisor, Jack Schwartz, for his support and encouragement. I am also grateful to Ed Schonberg and Robert Dewar for many interesting and helpful discussions, particularly during my early days at NYU. Terry Boult (of Lehigh University) and Richard Wallace have contributed materially to my later work on SETL through grants from the NSF and from ARPA. Finally, I am indebted to my parents, who gave me the strength and will to bring this labor of love to what I hope will be a propitious beginning.

Preface

Colin Broughton, a colleague in Edmonton, Canada, first made me aware of SETL in 1980, when he saw the heavy use I was making of associative tables in SPITBOL for data processing in a protein X-ray crystallography laboratory.

Accordingly, he loaned me B.J. Mailloux's copy of *A SETLB Primer* by Henry Mullish and Max Goldstein (1973). I must have spoken often of this language, because when another colleague, Mark Israel, visited the University of British Columbia two or three years later, and came across a tutorial entitled *The SETL Programming Language* by Rober Dewar (1979), he photocopied it in its entirety for me.

The syntactic treatment of maps in SETL places its expressive balance closer to algebraic mathematics than is customary for programming languages, and I immediately started finding SETL helpful as a notation for planning out the more difficult parts of programs destined to be coded in SPITBOL, Algol 68, Fortran, or Assembler/360.

When I took my M. Sc. in 1984–5, my enthusiasm about SETL was such that I made a presentation about it to my Advanced Programming Languages class. Later, when I was invited to stay on for my Ph.D. at the University of Toronto, the reason I declined to do so was specifically because no one there was willing to supervise work on SETL, and I for my part did not want to get caught up in either the logic programming or the (really very good) systems programming tradition prevailing there at that time.

So it was that in 1987, not having access to any good SETL implementation (the CIMS version seemed to crash on the first garbage collection attempt on the mainframe I was using), I decided to dash one off in SPITBOL. It was a good learning exercise. The compiler seemed to work correctly to the extent it was tested, and produced runnable SPITBOL code, but never saw much practical application on the hardware of the day.

In December of 1988, I decided to write a production-grade SETL compiler in C, and thus began an implementation that I use to this day and continue to extend. From the start, I found maps useful in the combinatorial programming I was doing in molecular modeling. The implementation is packaged in such a way that its most common invocation mode ("compile and go") is via the shell-level command setl, making SETL programs easy to fit into Unix filter pipelines, just like other scripts or pre-compiled programs.

Reasonably convenient though it was to be able to set up arrangements of communicating SETL programs in this way, the languages such as the Bourne and C shell used for interconnecting the various programs were still nothing more than thin descendants of the job control languages of yore. Since SETL was already so competent at handling data, it was only natural to extend it with facilities for process creation and communication. This led to the one-shot *filter*, unidirectional *pipe*, and bidirectional *pump* models for communication described in Chapter 2 of this dissertation.

The fact that a **writea** to a reader's **reada** can as easily transfer a large map as a small integer, combined with the fact that the Unix command rsh can be used to launch tasks on remote processors and communicate with them, made this a very comfortable programming environment for distributed programming, though not able to express a TCP/IP (TCP or UDP) server or client directly.

In 1994, the World Wide Web “arrived”, and it became clear that TCP/IP was to become firmly established as a global standard—the MIME-conveying HTTP protocol rides upon TCP streams, and the namespace-structuring Universal Resource Locator (URL) convention uses host names that map to IP addresses through the auspices of the widely used Domain Name Service (DNS). I therefore decided to build support for TCP/IP directly into the SETL I/O system, such that opening and using a bidirectional TCP communications stream in SETL would be as easy and natural as opening and using a file. How this is done is detailed in Chapter 3, which also describes SETL’s programmer-friendly support for UDP datagrams.

The ability to code servers in SETL has proven to be even more useful than I predicted. Servers act as the primary objects in server hierarchies. They bear state and control access to that state through message-passing protocols with child processes which in turn deal with clients. A server tends to keep track of such children with a dynamically varying map. Between the server and these trusted, proximal children, communication is safe and quick, minimizing the risk of the server becoming a bottleneck. The WEBeye study of Chapter 4 illustrates this pattern.

The liberal use of processes turns out to be beneficial time and again. Real work tends to fall to simple modules which communicate in a primitive way through their standard input and output channels, and these modules can easily be written in more efficiency-oriented languages than SETL where necessary. Small components are also easy to isolate for special or unusual testing, or for those rare but inevitable episodes called debugging.

Overall, systems designed as process-intensive server hierarchies tend to acquire a satisfying dataflow feel. Not only is this in the spirit of Unix filters, it also dovetails with SETL's value semantics, which abhor a pointer and cherish a copy, and in so doing avoid the hazard of distributed dangling references.

Contents

Chapter 1

Introduction

Public network services have to be coherent, reliable, and responsive in the face of errors, failures, attacks, and intermittent resource scarcity. SETL [181] turns out to be a convenient and powerful tool for dealing with this challenging environment. This dissertation describes extensions to SETL that are useful in data processing, especially when the Internet and numerous processes are involved. It is intended to serve as a tutorial on the design of moderately complex distributed systems using SETL, and accordingly provides many examples.

1.1 Why SETL?

First of all, SETL strives to put the needs of the programmer ahead of those of the machine, as is reflected in the automatic memory management, in the fact that flexible structures can be employed as easily as size-constrained ones can, and in the presence of an interface to powerful built-in datatypes through a concise and natural syntax. This high-level nature makes SETL a pleasure to use, and has long been appreciated outside the world of distributed data processing. Flexibility does not itself ensure good discipline, but is highly desirable for rapid prototyping. This fills an important need, because experimentation is a crucial early phase in the evolution of most large software systems, especially those featuring novel designs [135, 173, 66, 70, 71].

Second, SETL's strong bias in favor of "value semantics" facilitates the distribution of work and responsibility over multiple processes in the client-server setting. The absence of pointers eliminates a major nuisance in distributed systems design, namely the question of how to copy data structures which contain pointers. SETL realizes Hoare's ideal of programming without pointers [114].

Third, the fact that every SETL object, except for atoms and procedure values, can be converted to a string and back (with some slight loss of precision in the case of floating-point values), and indeed will be so converted when a sender's **writea** call is matched by a receiver's **reada**, means that SETL programs are little inconvenienced by process boundaries, while they enjoy the mutual protections attending private memories. Maps and tuples can represent all kinds of data structures in an immediate if undisciplined way, and the syntactic extension presented in Section 2.15, which allows record-style field selection on suitably domain-restricted maps to be made with a familiar dot notation, further abets the direct use of maps as objects in programs, complementing the ease with which they can be transmitted between programs. A similar freedom of notation exists in JavaScript, where associative arrays are identified with "properties" [152].

Fourth, strings themselves are first-class objects in SETL. They are completely flexible, such that assigning to a substring can change the length of the hosting string, just as a tuple can change length through subtuple assignment. Strings have a rich set of built-in operations for searching and manipulation based on algebraically formulated patterns. Further extensions to allow selections and substitutions to be specified by regular expressions in slicing notations and other forms are described in Section 2.14. Because strings are at the heart of data processing, it is vital to support them well, and SETL does.

Fifth, SETL's skill with general finite maps has welcome consequences for the data processing practitioner. At an abstract level, a data processing system can be viewed as a dynamic graph along whose arcs messages pass. The Khoros [134] system makes this abstract view concrete in its visual programming language, Cantata. The data in messages undergoes transformation and recombination in processing nodes, where maps represent data relationships directly. Processes of extraction tend to gather data around keys (often strings) that identify categories. Processes of association discovery correspond to set intersection, and merging is closely related to set union. The sets themselves are most often domains or ranges of maps, or sometimes projections of sets of more general tuples. Maps are so very much at the heart of SETL style that Dewar, in his 1979 book *The SETL Programming Language* [53], wrote:

> The general rule in SETL is to use maps wherever possible. This may take some practice, especially if you are used to programming in some other language, but remember this simple principle: find the maps, they are always there!

This is a principle that works well in practice. For example, if a process *P* has to multiplex input streams from several other processes, a map *M* over the corresponding I/O handles in *P* will often be used to track *P*'s state of knowledge about those processes, and the domain of *M* will be in the set of input handles that is passed to the **select** primitive (see Section 2.5) when *P* waits nondeterministically for I/O events.

Sixth, the absence of restrictions that are unhelpful to programmers brings with it a substantial measure of orthogonality and robustness. Orthogonality promotes the use of feature combinations that make sense, which is conducive to directness of expression. The absence of size restrictions similarly helps to eliminate clutter. For example, when programmers do not have to write extra code to deal with the fact that messages embedded in a TCP stream can be of arbitrary length, the most general case is handled gracefully and effortlessly. The importance of this for publicly exposed network servers cannot be overemphasized. If the only thing the most assiduous attack can do is bring down a subprocess *S* due to overall exhaustion of resources that are allocated to *S*, and *S* is the hostile client's only interface to the public service, then the damage is easily isolated. It does not even need to be damage *per se*, but can be naturally handled like an ordinary transaction that aborts when it cannot commit.

The remarkable adaptability of SETL and its gift for concise expression over a wide range of programming problems stem from its close connection to the foundations of mathematics. Set formers, modeled after set comprehensions, are a splendid case in point. They are highly accessible little pictures which encourage the programmer to take a dual view of sets as entities that can be characterized by predicates or constructed from parts. Tuple displays also exemplify SETL's directness of expression. In "fetch" contexts, they are enumerative denotations much like the written form of LISP lists, and in "store" positions, they show immediately the pattern of a required structure. Finally, SETL has freely borrowed the best ideas of other programming languages, such as the Algol family, APL [120, 169], and SNOBOL [99], as well as adding a few of its own.

The main significance of all the foregoing attributes of SETL for data processing over the Internet is that they pave the way for small programs. A small SETL program can do a lot, is not constrained by the usual obstacles to the communication of complex or pointer-bearing objects between processes, and is a well isolated module. Shared variables, the plague of concurrent programs, are simply not in the language, and are rarely missed: sharing resources is a serious matter that usually calls for a management mechanism that is best encapsulated in a module anyway. Namespace is adequately structured by a file system or URL-like convention for most purposes, but access to the objects in that space is best mediated by an appropriately synchronizing small process.

This raises the important issue of the data processing environment—an insular language may be admired, but it cannot scream. I have been fortunate in choosing to adopt the Posix [118, 119, 117] standards as a design benchmark for the SETL interface to files, commands, users, processes, and network communication. This operating system model has gained wide acceptance by vendors in the 1990s, and is now embodied in the X/Open specification commonly known as Unix 98 [154]. Chapters 2 and 3 are largely devoted to a presentation of those features in my current definition of SETL that employ and build upon this model.

1.2 A Brief History of SETL

SETL today is essentially the same language Jack Schwartz introduced 30 years ago in *Set Theory as a Language for Program Specification and Programming* [175]:

> 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.

The contrast between the expressive efficiency of mathematics and the obsessive parsimony of machine-oriented languages was highlighted in *On Programming* [177, p. vii]:

> 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 algo-

> rithms 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.

This perspective, as hinted in the first quotation above, sprang from the strong perception in the late 1960s that there was a need for a set-oriented language capable of expressing concisely the kind of set-intensive algorithm that kept arising in studies of compiler optimization, such as those by Allen, Cocke, Kennedy, and Schwartz [5, 43, 6, 41, 7, 10, 130, 11, 8, 9, 131, 178, 179, 180, 12, 132, 42]. *Programming Languages and their Compilers* [44], published early in 1970, devoted more than 200 pages to optimization algorithms. It included many of the now familiar techniques such as redundant code elimination and strength reduction, dealt extensively with graphs of control flow and their partitioning into “intervals”, and showed how to split nodes in an irreducible flow graph to obtain a reducible one. Many workers in the 1970s and 80s besides those just mentioned identified SETL, directly or indirectly, as a language whose implementation was greatly in need of solutions to difficult compiler optimization problems [80, 84, 85, 143, 82, 86, 123, 83, 148, 149, 174, 208, 3, 209]. SETL, while still far from the celestial sphere of pure mathematics, was nonetheless seen as occupying a very high orbit relative to other languages. It was SETL's distance from pure machines that made optimizing its implementations so important and at the same time so difficult.

The synergy between the study of code optimization and the high-level set language used for expressing optimization algorithms led to the SETL compiler project [186, 177], which was itself an abundant source of optimization problems. The SETL project produced, among other things, the SETL optimizer [178, 88, 77], a 24,000-line prototype written in SETL. Unfortunately, on the machines of the day, it was too large to apply to itself. This was a pity because not only is SETL a language which could benefit greatly from a good optimizer, it is also one whose semantic simplicity makes it particularly amenable to the flow-tracing techniques of machine-independent code optimization. The absence of pointers alone circumvents the issue of aliasing, a huge advantage in this kind of analysis.

The sort of data flow (definition-use) information obtainable from analysis of control flow graphs, and more generally from Schwartz's "value flow" tracing [177, 178, 144] that could follow objects when they were stored in aggregates and later extracted, was useful in all sorts of ways. It sustained copy optimization [175, 178, 179], where the redundant copying of an object could be suppressed when the only subsequent use of the object also modified it, perhaps incrementally. Value flow analysis provided a dependency framework wherein the types of many variables and expressions could be deduced by a transitive closure process starting from the manifest types of literals and other forms [196]. This typefinding process in turn enabled the discovery of relationships of set membership and inclusion [180], which was itself a prelude to automatic data structure choice, because the way an object is used has a profound influence on how it should be implemented. Weiss and Schonberg [208, 209] later showed how to do type inference even in the presence of infinite sets of possible types arising from actions such as "$x := \{x\}$".

Data structure representations had their own sublanguage, the DSRL, which served to annotate, but not otherwise modify, SETL programs coded at an appropriately abstract level. The DSRL was designed to permit a smooth transition from Schwartz's two-level programming regime, in which programmers supplied representational details, to a more fully developed system in which a sophisticated optimizer made the selections [175, 56, 174, 88, 77]. An important concept in the DSRL was that of *base sets*, which were implicitly defined objects that could in principle allow much representational sharing among the objects conceived by the programmer.

Value flow analysis, type inference, copy optimization, and deeper determinations of relationships such as set membership or inclusion between variables preparatory to automatic data structure selection all embody an approach to program analysis described by Sintzoff [189] and called *abstract interpretation* by Cousot and Cousot [47] or *symbolic execution* in Muchnick and Jones [149, p. xv]. The essence of this model is that any program P with well-defined semantics can be projected onto a more abstract program *A* capturing salient properties of objects in *P* in a manner susceptible of analysis. For example, the sign of a product can be deduced from the signs of its multiplicands without knowing their specific values. Similarly, result types for known operators can usually be gleaned at compile time from operand types regardless of the actual run-time values of those operands. In abstract interpretation, the abstract program *A* is exercised at *P*'s compile time to discover desired properties of objects in *P*. The symbols in *A* combine and recombine according to an algebra appropriate for their purpose. If that algebra has been designed with feasible goals in mind, the exercise will converge. It is typical to ensure this termination by taking advantage of the fact that any set generated by inductive definitions (such as data flow equations) can be defined as the lattice-theoretic least fixed point of a monotone function. This often allows global properties to be inferred from local ones by a straightforward process of transitive closure.

The power and generality of abstract interpretation moved Paige and his colleagues to undertake an ambitious study of program *transformations*, which ultimately led to the APTS project [33, 161, 126, 162]. The first of the main three transformations used in APTS is *dominated convergence* [32] for computing fixed points of Tarski [195, 48] sequences ($f^i(1)\,:\,i =$ 0, 1, 2, ... for deflationary, monotone f) with reasonable efficiency. The second is *finite differencing* [157, 164, 158], which is a set-theoretic analogue of strength reduction that allows some expensive set operations within loops to be reduced to incremental updates by locating fixed points more quickly through the construction and maintenance of program invariants. The third transformation is *real-time simulation* [159, 30, 160, 34, 166] of an associative memory on an ordinary random-access memory (or with slight additional restrictions a mere pointer-access memory), which effectively automates the tedious programming activity of choosing efficient basings for sets.

Chung Yung has recently used finite differencing in a technique he calls *destructive effect analysis*, which seeks to incrementalize the copying of aggregates, in his purely functional programming language, *EAS*, a packaging of the typed λ-calculus as extended with homogeneous sets [210, 211, 212].

Transformational programming can be regarded as a formalization of Dijkstra's *stepwise refinement* [57, 58]. As Bloom and Paige [28] point out, the transformational methodology is able to do much more than merely optimize code, or translate a SETL-like language into a C-like one. By helping the algorithm designer reason about time and space complexity in syntactic terms rather than only by means of low-level counting arguments, this technology has actually played a significant role in the invention of several new algorithms with greatly reduced asymptotic complexity compared to previous solutions [165, 163, 32, 31, 28, 34, 38, 93], while it has rendered the algorithms themselves more perspicuous both to their inventors and to their students.

The next phase in the development of APTS will seek to improve both its reliability and its performance. Currently, all program transformations in APTS are proved correct (meaning-preserving) by hand, which is slow and error-prone. The hope is to integrate a meta-level proof verifier along the lines of Etna [36], an outgrowth of Cantone, Ferro, and Omodeo's work on fast decision procedures for fragments of finite set theory [37].

Alternatively, the model for an integrated verifier might be the SETL-like NAP [126] system, itself implemented in the SETL derivative Cantor [127, 124]. Verification of assertions in, say, Hoare logic [113] would increase confidence in automatically applied transformations. Davis and Schwartz [50] showed how mechanical verification systems could extend themselves with new proof methods without violating soundness or changing the set of statements that could be proved.

The main existing impediment to the speed of APTS is the fact that its database of program property relationships, which is dynamically deduced using a static database of inference rules, must be recomputed after each application of a program transformation. What is being sought is an *incremental* rule database system that can be used to regenerate the relationship records efficiently after each rewriting operation [161]. Ultimately, it should be possible to apply APTS to itself for further large gains in speed [162], and to take advantage of the technique of partial evaluation [122] to realize a production-grade transformational system.

Recently, Goyal and Paige [94] have revisited the copy optimization problem for SETL and other high-level languages that exemplify Hoare's ideal of a pointer-free style of programming. By taking the well-known technique of dynamic reference counting to achieve "lazy" copying, and combining that with static liveness determination based on Schwartz's value flow analysis, they are able to optimize the placement of copy operations and of **om** assignments, the latter serving to decrement the reference counts of objects known to have no subsequent uses. They also prove the correctness of their alias propagation analysis and code transformations using formal semantics and abstract interpretation.

Goyal [91] has obtained a dramatic improvement in the algorithmic complexity of computing intra-procedural *may-alias* relations, again using dominated convergence and finite differencing. In his dissertation [92], he develops set-theoretic languages which can express both abstract specifications and low-level implementations in a form which uses a data structure selection method based on a novel type system to preserve the *computational transparency* that is necessary in order for statements about program efficiency to be meaningful. This is a cornerstone of the general transformational methodology.

There have been a number of implementations of SETL and SETL-like languages over the years. The first was called SETLB [151] and was implemented using an extension of Harrison's extensible LISP-like BALM language, BALMSETL [101, 187]. SETLB was succeeded by SETLA [176], was implemented in BALMSETL, and was an almost strict subset of SETL.

The full SETL language itself was implemented in LITTLE [207], syntactically a Fortran-like language supplemented with a notation for bit-field extraction. LITTLE had only two built-in data types, fixed-length bit strings and floating-point numbers, but was used to implement both the compiler and run-time system of the version of SETL maintained and distributed by the Courant Institute of Mathematical Sciences (CIMS) at New York University from the mid-1970s until the late 1980s.

The CIMS SETL system was quite slow and cumbersome, and LITTLE was not widely ported, so in the late 1970s Nigel Chapman, who was then a graduate student at the University of Leeds, designed and implemented a system called Setl-s [39]. It covers a substantial subset of SETL, leaving out only such ephemera as the macros, backtracking, the data representation sublanguage, support for separate compilation, and a few minor syntactic luxuries. The "-s" in the name can also stand for "small", because Setl-s was a very compact system based on Dewar's celebrated *indirect threaded code* [55] technique, and was written in MINIMAL, the portable assembly language in which the run-time system of MACRO SPITBOL [54] was implemented. Jay VandeKopple dropped the hyphen from Setl-s and worked on what was then called SETLS in the 1990s [202] while at NYU on sabbatical from Marymount College. He maintains the current version of the compiler and documentation [203].

The batch-oriented character of the full CIMS SETL implementation, its requirement for considerable computing resources, and to some extent its complexity and slowness, led Ed Dubinsky, who in the early 1980s was using SETL informally for discrete mathematics and abstract algebra courses at Clarkson University, to collaborate with Gary Levin in the mid-1980s on the creation of a new, small, interactive system close in syntax and spirit to SETL but without the overhead. Called ISETL [145], this system has now been stable at version 3.0 for many years, is freely and publicly available, comprises little more than 20,000 lines of portable C code, runs quite responsively on all common desktop computers, provides first-class functions (a feature Dubinsky has long valued), and has strong ongoing pedagogical support through several textbooks, annual workshops, and the enthusiasm of a sizeable community of mathematics teachers [22, 150, 78, 183, 23, 79, 76, 81].

In the late 1980s at NYU, meanwhile, there was a project aimed at overhauling the CIMS SETL language and implementation. The new version of the language was tentatively named SETL2. Kirk Snyder, a graduate student at that time, became dissatisfied with what appeared to be ceaseless discussion supported by little action, and covertly designed and implemented his own system called SETL2 [190] in about a year. It simplified and modified various aspects of SETL syntax and semantics while it removed the usual apocrypha (macros, backtracking, the data representation sublanguage, and SETL modules), and introduced Ada-based "packages" and a portable file format for separate compilation on DOS, Macintosh, Unix, and other platforms. Snyder subsequently added lambda expressions (first-class functions with closures) and support for object-oriented programming, including multiple inheritance [191], though these extensions were not entirely without semantic problems in the context of a nominally value-oriented language. Recently, Toto Paxia has made improvements to the interoperability of SETL2 through his "native" package declarations [168] which allow more direct calls out to routines written in C than were previously possible.

Perhaps the most famous of all SETL programs to date is Ada/Ed [2, 135, 173], the first validated translator and executable semantic model for the language now known as Ada 83 [198]. It established convincingly that SETL is well suited for the rapid prototyping of complex systems of significant size, and that when care is taken in the construction of such prototypes, they can serve as readable, definitive specifications to inform and guide the building of production systems.

SETL's success as a prototyping tool spawned the Esprit SED (SETL Experimentation and Demonstration) project [125] of the late 1980s, which was a sweeping effort to create a SETL-based prototyping environment complete with highly sophisticated language manipulation tools at the syntactic and semantic levels [73]. This included a SETL-to-Ada translator [72, 69], an editor, a debugger, and a performance-profiling monitor [29]. The latter was rendered particularly accommodating and non-invasive by the use of coöperating processes sharing messages over TCP sockets. Abstract interpretation was the operative model in both the ambitious Tenenbaum-based [196] type inferencer and Paige's more general RAPTS [158] transformational system (the predecessor to APTS), which was used to prototype "meta-SETL" [4], an AST-traversing interpreter. SED employed a rich set of language processing tools such as Typol [51, 52] for type checking and other semantic analysis via pattern-directed inference (abstract interpretation), and a Mentor-based [74] interface to the syntax-directed editing environment. Interoperability was addressed in the SETL-to-Ada translator and in the performance monitor by means of ISLE (the Interface Specification Language and Environment), which was important for the SED project's demonstration of rapid prototyping in SETL in a cartography application containing a package of computational geometry algorithms [27].

Jean-Pierre Keller, the leader of the SED project, went on to define a SETL descendant which he called Cantor [127, 124]. Cantor is actually closer in syntax and semantics to ISETL [145] than to SETL, and is implemented in ISETL. It has first-class functions, some concurrency mechanisms, and a set of predefined objects for

GUI construction.

Encouraged by SED's contributions to the art of programming in the large and by the project's inchoate plans for persistence in SETL, but disappointed by SED's failure to arrive at a coherent product [70], Ernst-Erich Doberkat proposed integrating persistent backing stores called *P-files* and their requisite namespace support into SETL/E [66, 61], a revision of SETL that was extended with a process creation operator and renamed ProSet [71] to signify its role in prototyping. Doberkat's interest in SETL during the 1980s [59, 65, 63, 72, 68, 69] grew in the 1990s into a more general interest in software engineering with set-oriented languages having intrinsic persistence features [60, 64, 67, 70, 71, 62] that sought to spare the programmer the trouble of coding data movement operations explicitly. Willi Hasselbring also showed how to translate [103] a subset of SETL/E into Snyder's SETL2. ProSet tuples are natural candidates for inter-process communication via Linda tuple spaces [90], so Hasselbring has worked extensively throughout the 1990s on and with a methodology for prototyping concurrent applications using a hybrid system called ProSet-Linda [102, 104, 109, 105, 106, 107, 108, 111], and compared this approach to several others [110, 112].

An entirely different notion of persistence, pertaining not to a backing store but to the ability of a data structure to retain its update history in a way which preserves the time and space efficiency of access to current and past states [75], was used by Zhiqing Liu to create a SETL run-time system and graphical debugging interface which allows users to scroll backward and forward in a program execution history [138, 140, 139]. Liu has also tried to bring some of the convenience of SETL into the relatively low-level world of C++ with his LIBSETL [141] header files and run-time library.

Another relative of SETL is Slim [20], which its designer, Herman Venter, describes as “more like a cousin to SETL than a child, since it shares a common heritage with SETL, but was independently designed” [204]. It supports object-oriented programming and allows optional type declarations. Both the language and its implementation are relatively small but complete.

An experimental language which bears some kinship to SETL at the data structure level is the functional language SequenceL [45, 46], formerly called BagL. Every data item in SequenceL is a sequence—even single elements are viewed as one-element sequences. This facilitates a highly orthogonal treatment of operations and particularly distributivity, rather in the spirit of APL. SequenceL also has maps for use in associative subscripting. This language provides few syntactic comforts, however, offering little more than a few denotational forms and a prefix function application notation that is used even for binary operators.

The Griffin [95] language which was designed at New York University in the early 1990s was intended to be a general-purpose successor to SETL. Its goals were very lofty, and included what is surely the most comprehensive type and type inference system ever proposed. Griffin was supposed to give the programmer complete freedom of choice as to whether to code in a functional or an imperative style. It had language constructs for database-style transactions, namespaces, and persistence. Real-time facilities, Ada interoperability, exceptions, and broad support for concurrency were also built in at the language level. Much of Griffin has been implemented in a compiler, but a major obstacle to the completion of that enormous task has been the difficulty of fixing on a fully self-consistent language definition.

My own interest in SETL in the 1980s, and my dissatisfaction with the CIMS implementation and the terms under which it was marketed, led me in 1987 to prototype a compiler and run-time interpreter for SETL in SPITBOL [54]. Although this version was a reasonably complete realization of the core SETL language as described by Dewar [53], and appeared to function correctly in the limited tests to which it was put, it was entirely unsatisfactory in terms of speed, and had size limitations on strings, tuples, and sets imposed by their rather direct representation as SPITBOL strings, arrays, and tables respectively.

So it was that in late 1988, newly deprived of SPITBOL in the transition from a mainframe running MTS (the Michigan Terminal System [188]) to a workstation running Unix, I found myself encouraged by the emergence of ISETL but unable to adapt it easily to non-interactive data processing purposes. I thus began a part-time effort to implement a compiler and run-time system for SETL in C. This version was complete enough for daily use by mid-1990, and, though it is now rather like the car in which almost every part has been replaced, it is still my main vehicle for running SETL programs. It is also the one which supports the language extensions and code samples presented in this dissertation. Although I set much greater store by robustness and correctness than by speed in this implementation, its efficiency has almost always been satisfactory. In the rare instances where this has not been the case, it has been easy to make the SETL program interoperate with programs or subroutines written in lower-level languages, and in any case those situations usually seem to occur when there is a library of C functions already involved. Elementary computation-intensive image processing operations are a representative example.

Shortly before arriving at New York University in 1990, I became aware of Snyder's SETL2, which had just been unveiled that spring. In the fall of that year, there was some talk of a unified language definition and implementation, but this was prevented by the fact that Snyder did not want to permit access to his system's source code, and by the fact that we both considered SETL and SETL2 to be works in progress. There were also some differences between the two languages, and they were very differently packaged, reflecting their rather different goals. Since there was starting to emerge a body of SETL2 code, including a revision of Ada/Ed [21], I elected to extend the SETL grammar with as much of SETL2's generally simpler syntax as it could accommodate. It quickly became apparent that essentially nothing needed to be left out. The worst collision was that some ambiguities in the syntax of loop headers were a little awkward to resolve, but this only led to some minor patchwork in the compiler and scarcely incommoded the SETL programmer. The resulting augmented SETL is a less than ideal splice from the language design point of view, but perfectly comfortable from the standpoint of writing code except where it gratuitously requires the programmer to make occasional choices between almost equivalent alternatives. This slight surfeit of form does not appear to have any negative impact on the readability of SETL programs, however, and in fact if the programmer always chooses the simplest expression available, it arguably even confers a minor advantage. For example, most loops can begin with the comparatively simple SETL2 headers, but there are times when the greater generality of the full SETL loop construct is preferable, particularly when the exit test does not fall naturally at the very beginning or end of the loop.

Throughout most of the 1990s, I have continued to work on and with SETL. From 1991 on, I have deliberately been rather conservative about extending the syntax of a language I already consider to be among the world's finest, but have not been so hesitant about its run-time system. My goals are highly pragmatic, as I am a day-to-day user of SETL as well as a zealous apologist.

Set-theoretic languages have a small though active following, particularly in the logic and functional programming communities. For example, SPARCL is billed as "a visual logic programming language based on sets" [192], and is essentially a graphical shell for a version of Prolog [40] augmented with sets, where constraints on the *partitioning* of the sets give the language much of its expressive power. Escher [142], on the other hand, descends from the functional language Haskell [197], and extends the very general and useful mechanism of list pattern matching on function signatures to accommodate sets, which are themselves identified with predicates. Evaluation in Escher is based on pure rewriting, and although it has no unification built in to its computational model, the pattern matching and lazy evaluation that are hallmarks of Haskell, together with an added ability to reduce expressions containing variables, combine to support logic programming in Escher without sacrificing the advantages of the functional style. Two Web sites [170, 172] are devoted to various aspects of programming with sets, and twelve papers were presented at a recent workshop in Paris on *declarative* programming with sets [171]. Declaring goals in preference to specifying operational steps is again a subject that is close to the heart of logic programming.

1.3 Summary

This chapter has presented some of the general background and motivation of SETL, and suggested why it remains a language worthy of further study and development.

Since data processing is largely concerned with the interaction of programs with their environments, the next chapter examines the rich repertoire of input/output and related facilities that have been added to SETL. Subsequent chapters deal with the more network-specific provisions, and illustrate the design patterns I have found most effective in the use of these new tools.

Chapter 2

Environmentally Friendly I/O

The CIMS version of SETL as described by Schwartz et al. [181] already had many features and characteristics that made it useful as a data processing language, both in general because of its high-level nature and in particular because of such extensions as the SNOBOL-inspired string pattern matching routines.

However, the implementation was not packaged in the convenient manner of popular Unix tools such as awk[1] or perl, which promote the construction of simple programs that can be chained, output to input, with other such programs to build up pipelines of coöperating processes. Also, while the language specification and the the pre-1990 CIMS implementation provided for sequential file- and printer-oriented I/O, it offered no built-in way to communicate with or spawn external processes.

[1]Unix 98 [154] functions and commands are not individually cited throughout this dissertation, but are indicated by a suggestive typeface. This convention is also used for common GNU [87] utilities such as fmt and perl.

SETL2 added support for direct-access files [191] and a **system** primitive [190] able to start an external program and wait for it to complete, but still did not offer any way to communicate with external programs except by way of files.

Distributed data processing depends in a direct and fundamental way upon good I/O facilities, however, and this chapter describes various extensions to SETL relating to inter-process communication, including such matters as string and number formatting, multiplexed I/O, timers, signals, and the Unix 98 interface [154]. Chapter 3 deals with network-specific extensions. In non-Unix systems, it is expected that where a particular SETL feature is not easily mapped to a corresponding facility of the operating system, a SETL implementation will supply a benign substitute or balk as appropriate.

2.1 Invocation Environment

Following SETL2, which adopted the identifiers conventionally used in Ada to refer to the external name and arguments with which the program (the command) was invoked,

command_name

is a predefined constant string that represents, in some environment-defined way, the name of the program from the point of view of the party that launched it, and

command_line

is a predefined constant tuple of strings representing a list of arguments to the program.

Whenever one of the commonly used Unix shells is the caller, **command_name** and **command_line** correspond to what C programs receive as argv[0] and the subsequent argc−1 elements of argv respectively.

If the setl command is invoked directly, then **command_line** will be a tuple of strings representing the arguments following "--" or "-x", and **command_name** will be the name of the SETL interpreter, setlrun in the default configuration of my current SETL implementation [19].

If, however, the SETL interpreter is invoked via the "#!" escape common to virtually all varieties of Unix (though this escape sequence is of unspecified effect according to the Unix 98 standard), by placing

```
#! /path/to/setl -k
```

at the very beginning of a file that otherwise contains a SETL program, and by making this file executable using chmod (see Section 2.7), then the filename will be available inside the SETL program as **command_name** and any arguments placed after that name will be the strings that show up in **command_line**.

A convenient way of supplying some environmental information to programs is through environment variables. All widely used operating systems support these in one form or another. In SETL,

value := **getenv** *name*;

is used to retrieve, as a **string**, the value associated with the environment variable named *name*. If no such variable is known to the environment, **getenv** returns **om**.

Programmers should be aware that the names of environment variables may or may not be treated case-sensitively, depending on the system. In Unix systems, case is significant; in DOS-based systems, it is not. I strongly recommend strict adherence to uppercase as a convention for these names, since that is a portable practice that is already well entrenched in the Unix community.

A SETL program can also set environment variables for itself and for the benefit of programs it spawns, using

setenv (*name*, *value*); -- associate *value* with *name*

where *name* and *value* are both strings, and *value* defaults to the null string. Finally, environment variables can be deleted:

unsetenv (*name*);

2.2 Filters, Pipes, and Pumps

One of the great contributions of Unix to the world of data processing was its tendency to encourage the use of small, modular processes that can be connected together in *pipelines*, also known as *filter chains*, where the standard (default) output stream of one process would feed into the standard input of another, the whole chain thus forming a larger module whose standard input and output would be available for further redirection to any file, device, or process. Programs designed to act as pipeline elements are naturally called *filters*. Filtering is usually a one-shot event in the sense that a filter will typically read all its input and then quit, having produced some output along the way. All Unix shells in popular use have the same basic syntax for connecting filters into pipelines: a vertical bar between two simple command (program) invocations indicates that the standard output of the first is to be passed into the standard input of the second. For example,

```
cat *.txt | fmt –60 | wc
```

concatenates all files with a .txt suffix in the current directory into a stream which is fed into a command, fmt, that treats its input as text to be left-justified in paragraphs not to exceed 60 characters in width. The output of fmt in turn passes into wc, whose output is simply a report of the number of lines, words, and characters in its input. If this pipeline had been typed in at an interactive text command shell, with its output left attached to the display, the output of wc, a single line in this case, would simply be displayed as text looking something like this:

```
89 378 1429
```

It is also possible for programs to engage other programs as child processes with communication arrangements such that the parent's I/O handle is connected to the standard input and/or standard output of the child. The unidirectional channels, called *pipes*, are typically used by programs to read or write some data through a filter in situations where it is convenient for the parent to execute several I/O operations in the course of reading or writing the data.

The bidirectional case is the *pump*[2] stream, where a program's I/O handle is connected to both the standard input and standard output of the child process. There is generally some sort of protocol involved in this case, and the parent-child interaction will often span considerable real time. We will see many examples of this in Chapter 3, where processes called *servers* deal with clients only through child processes, each of which normally exists for the duration of a client connection. Pump streams are also useful outside the context of networks, as for example when a program spawns local GUI (graphical user interface) processes.

2.2.1 Filters

Because SETL is good at handling strings, it is convenient to use it both for processing strings and for passing them to other programs. The SETL statement

$$\mathit{output} := \textbf{filter}\ (\mathit{cmd}, \mathit{input});$$

causes *cmd* to be submitted to the standard Unix 98 [154] "shell" command language interpreter, `sh`, through its `-c` (command) argument. The command specified by *cmd*, which may internally contain pipeline and other I/O redirection indicators, is run in a child process. The string *input*, which defaults to the null string, is fed into this child's standard input, and the string *output* receives everything that issues from its standard output stream.

[2] *Pump* is a term I have used for many years. Unix has never really embraced the notion of a bidirectional buffered stream.

If **filter** is unable to create a child process due to resource exhaustion, it returns **om**. When the string *input* is non-null, two child processes may need to be created: one to run the command *cmd*, and one to feed *input* into the command. The parent SETL process remains as the "consumer" that builds a string containing the command's output, to be returned to the caller of **filter**.

The following is a simple example of the use of **filter** to format and left-justify text so that it fits within a prescribed width such as might be imposed by a user's text window. The program in Section A.33 uses this technique. This subroutine runs an external command, fmt, to insert end-of-line characters in the appropriate places:

```
proc fill_message (text, width);          -- wrap text
  return filter ('fmt -' + str width, text);
end proc;
```

In this example, **str** is used to convert the presumed positive integer parameter *width* to a decimal string, which is appended to 'fmt -' to form the whole command including the command-line parameter. The string *text* is filtered through fmt, and the formatted result is returned.

2.2.2 Pipes

A unidirectional stream connected to the standard input or standard output of an child process is called a *pipe*.

In SETL, here is how to start an external command as a child process and open an input pipe stream connected to its standard output:

fd := **open** (*cmd*, ‘pipe-from’); -- or ‘pipe-in’

To launch an external process with an output pipe stream connected to its standard input, use

fd := **open** (*cmd*, ‘pipe-to’); -- or ‘pipe-out’

In both of these cases, *cmd* can be any string that makes sense to the environmental command interpreter (the shell), just as for **filter**.

The stream handle returned by **open** in the above prototypes is assigned to the variable *fd*, as a mnemonic for *file descriptor*. The SETL programmer should treat it as opaque and certainly never do arithmetic on it, but may wish to be aware, especially when setting up communication with programs written in languages other than SETL, that the SETL file descriptor is exactly the integer that is assigned by the kernel as the result of open and related calls, and is called a file descriptor throughout the Unix literature. SETL implementations are expected to provide buffering over this handle, as detailed in Section 2.2.4. If **open** fails to create a child process due to resource exhaustion, then it returns **om** instead of a valid file descriptor.

Below is an example of the use of ‘pipe-from’, where an input pipe stream is connected to the standard output of the Unix ls command to obtain a list of files in the current working directory, one filename per line. To each filename read from the ls process, the SETL program applies the **fsize** operator to discover the size of the named file in bytes, and prints the resulting integer right-justified in a 10-character field beside the left-justified filename, separated by a space:

```
fd := open ('ls', 'pipe-from');                  -- open file-listing subprocess
while (name := getline fd) ≠ om loop             -- loop for each input name
  print (whole (fsize name, 10), name);          -- print file size and name
end loop;
close (fd);                                      -- close child process
```

Here is an example of 'pipe-to', where the SETL program opens a stream to a print spooler, lpr:

```
log_fd := open ('lpr', 'pipe-to');
printa (log_fd, 'Log begins at', date);
```

There are also primitives named **pipe_from_child** and **pipe_to_child** which are essentially degenerate forms of the **pump** primitive described in Section 2.2.3.

2.2.3 Pumps

An external command can be started as a child process with its standard input *and* output connected to a bidirectional stream in the parent SETL program as follows:

```
fd := open (cmd, 'pump');
```

Even without the direct appearance of sockets, this is a powerful tool for distributed computing, because the string *cmd* can specify an invocation of rsh to execute, for example, the spitbol command on a remote host even if the local one doesn't have spitbol executably installed, or wishes to distribute its load.

Sometimes, instead of starting an external command as the specification of a child process, it is convenient to create the child as a clone of the currently executing SETL program. The new nullary primitive **pump** creates a child which inherits a copy of the parent SETL program's data space in the manner of **fork** (see Section 2.17.2). If successful, it returns -1 in the child and returns a bidirectional file descriptor in the parent, connected to the standard input and output of the child. If unsuccessful due to resource exhaustion, it returns **om**, as does the 'pump' mode of **open** above:

```
fd := pump();   -- the optional "()" suggests more than a mere fetch
```

This fragmentary code template shows how to use **pump**:

```
fd := pump();   -- spawn clone
if fd = −1 then
  -- Child:  I/O on stdin, stdout, which are connected to parent's fd
    ⋮
  stop;   -- normal exit
end if;
if fd ≠ om then
  -- Parent:  I/O on fd until EOF tells us the child has completed
    ⋮
  close (fd);   -- clear child from process table
else
  -- Child process could not be created—handle or ignore failure
    ⋮
end if;
```

The reason it usually works best to put the child code first is that it is a program in miniature, exiting just before the end of the block that contains it, whereas the parent is most likely to save the new *fd* and carry on. It would be clumsy to have the parent's code section end with a branch around the child's code. An exception to this rule is where that branch is really a **return**, as in this generic launcher:

```
proc start_helper (helper);     -- launch helper and return its pump fd
 fd := pump();                  -- spawn clone
 if fd = om then
  -- No child created
  return om;                    -- failure return
 elseif fd ≠ −1 then
  -- Parent process, with fd connected to child
  return fd;                    -- caller will use and then close fd
 end if;
 -- Child process, with stdin and stdout connected to parent
 call (helper);                 -- indirect call to the helper procedure
 stop;                          -- guard against helper neglecting to exit
end proc;
```

One program which uses **pump** is the second version of impatient.setl in Section 3.3.1.

2.2.4 Buffering

For output to files, print spoolers, and one-shot filters, the SETL programmer may never need to be aware of buffering, but when processes are interconnected through pipes and pumps, there are times when it will be necessary to tell the I/O system explicitly to move all data currently accumulated in a stream buffer out to the receiver. This is done by the following call:

```
flush (fd);                -- get the kernel caught up
```

One feature of the pump stream, whether created by **pump** or by **open** specifying mode ‘pump’, is that its output side is automatically flushed whenever a read from its input side is attempted. In fact, this is true for all bidirectional and direct-access streams created by SETL, such as the ones listed in Section 2.3 and the socket streams introduced in Chapter 3.

This automatic flushing association between the input and output sides of a bidirectional stream is called *tying*, and can also be requested between any otherwise independent pair of streams where one is open for input and the other for output, using the call:

tie (*fd_in*, *fd_out*); -- autoflush *fd_out* on each *fd_in* input try

Thus it is common to see the statement

tie (**stdin**, **stdout**); -- autoflush **stdout** on each **stdin** input try

near the beginning of SETL programs intended to be invoked through the 'pump' mode of **open**. This association is made automatically in the child process arising from a successful **pump** call. A program intended merely as a filter, by contrast, will *not* tie **stdin** to **stdout** if it wishes to operate line by line internally and yet remain buffer-efficient.

Buffering is not part of the SETL language specification, but implementations are expected to make the behavior as much like that of the FILE type in the C stdio library as possible. By default, **stderr** should be flushed after every character, and other output streams should be "block buffered" (meaning only automatically flushed when the buffer fills up) except when connected to a terminal-like device, in which case they should be "line buffered" (flushed at least after each output line).

2.2.5 Line-Pumps

There is one more variant of the versatile pump stream, available through the I/O mode 'line-pump':

fd := **open** (*cmd*, 'line-pump');   -- or 'tty-pump'

The difference between a *line pump* and a regular pump is that the environment provided to the child process in the case of the line pump is as much as possible like a line-by-line virtual terminal. Many programs, including the usual Unix shells, govern their behavior according to whether the output model is another program or a user at such a terminal.

Most significantly, the standard C stdio library uses line buffering (in the sense described in Section 2.2.4) instead of block buffering on the standard output stream when it is connected to a line-by-line terminal, and programs rarely change this default. Hence it is possible to use many "off-the-shelf" programs as coöperating child processes even when they were intended as filters, at the cost of assuming something about each such program's implementation, specifically its output flushing policy.

The line pump is a rather specialized feature and probably best avoided in code intended to be ported easily outside the Unix world, but it can be very handy for setting up automated interactions with programs such as mail clients. For example, I was easily able to expunge several thousand unwanted mail messages that an out-of-control robot recently sent me, by the simple expedient of having a small SETL program invoke the Unix Mail client program and "type" the deletion command in response to each message it recognized as being from the robot.

2.3 Sequential and Direct-Access I/O

Apart from the I/O modes which create "sockets" as detailed in Chapter 3, the pipe and pump modes of Section 2.2, and the signal and timer modes described in Section 2.4, the choices of second ("mode") parameter to **open** (some of which have appeared in previous versions of SETL) are as follows. There are many synonyms here, largely for the sake of backward compatibility. There is no distinction between "binary" and other modes. Translation of end-of-line sequences on non-Unix systems is expected to be done by external filters if necessary, and the older meaning of binary I/O as input or output of some machine-level representation of SETL values is obsolete—the efficiency advantage of such modes is negligible, and the inconvenience significant. Moreover, SETL strings can accommodate any bit pattern that might be required by a foreign data format, so the coverage of needs in this regard is complete:

MODE	SYNONYMS	MEANING
'r'	'rb', 'input', 'text', 'text-in', 'coded', 'coded-in', 'binary', 'binary-in'	stream input
'w'	'wb', 'output', 'print', 'text-out', 'coded-out', 'binary-out'	stream output
'a'	'ab', 'append', 'output-append', 'print-append', 'text-append', 'coded-append', 'binary-append'	stream output starting at end of file

'n'	'nb', 'new', 'text-new', 'new-text', 'coded-new', 'new-coded', 'binary-new', 'new-binary'	stream output to new file
'rw'	'read-write', 'input-output', 'twoway', 'two-way', 'bidirectional'	bidirectional stream
'r+'	'rb+', 'r+b', 'direct', 'random'	direct access file
'w+'	'wb+', 'w+b'	empty file and then do direct access
'a+'	'ab+', 'a+b'	direct access file, always write at end
'n+'	'nb+', 'n+b', 'new+', 'new-r+', 'new-w+', 'direct-new', 'new-direct', 'random-new', 'new-random'	direct access to new file

2.3.1 Sequential Reading and Writing

Most of the names of the sequential I/O routines in the current version of SETL have appeared in previous versions, but there are a few new ones, and the semantics of some of them are slightly different from both CIMS SETL and SETL2 (which of course differ from each other):

get $(line_1, line_2, \ldots)$;	-- **geta** from **stdin**
geta $(fd, line_1, line_2, \ldots)$;	-- $line_1, line_2, \ldots$ are **wr** args
getb $(fd, lhs_1, lhs_2, \ldots)$;	-- $lhs_1, lhs_2, \ldots$ are **wr** args
c := **getc** fd;	-- retrieve one character
c := **getchar** $()$;	-- **getc** from **stdin**, "()" optional
s := **getfile** fd;	-- all characters up to EOF
s := **getline** fd;	-- one input line
s := **getn** (fd, n);	-- up to n characters

c := **peekc** *fd*;	-- look ahead one character
c := **peekchar** ();	-- **peekc** from **stdin**, “()” optional
print (*rhs*$_1$, *rhs*$_2$, ...);	-- **printa** to **stdout**
printa (*fd*, *rhs*$_1$, *rhs*$_2$, ...);	-- space-separated values on a line
nprint (*rhs*$_1$, *rhs*$_2$, ...);	-- **print** without line terminator
nprinta (*fd*, *rhs*$_1$, *rhs*$_2$, ...);	-- **printa** without line terminator
put (*line*$_1$, *line*$_2$, ...);	-- **puta** to **stdout**
puta (*fd*, *line*$_1$, *line*$_2$, ...);	-- *line*$_1$, *line*$_2$, ... are **rd** args
putb (*fd*, *rhs*$_1$, *rhs*$_2$, ...);	-- *rhs*$_1$, *rhs*$_2$, ... are **rd** args
putc (*fd*, *s*);	-- *s* is any string
putchar (*s*);	-- **putc** to **stdout**
putline (*fd*, *line*$_1$, *line*$_2$, ...);	-- same as **puta**
putfile (*fd*, *s*);	-- similar to **putc**
read (*lhs*$_1$, *lhs*$_2$, ...);	-- **reada** from **stdin**
reada (*fd*, *lhs*$_1$, *lhs*$_2$, ...);	-- **read** and convert values
write (*rhs*$_1$, *rhs*$_2$, ...);	-- **writea** to **stdout**
writea (*fd*, *rhs*$_1$, *rhs*$_2$, ...);	-- same as **putb**

Print, **printa**, **putb**, **write**, and **writea** take **rd** arguments of any type, and write strings on a single line, with spaces between the represented values. **Nprint** and **nprinta** are just like **print** and **printa** respectively, but leave the output line unterminated. **Putb** is functionally identical to **writea**, and converts values the same way **str** does. **Print** and **printa**, however, treat strings as a special case, and leave them unquoted. **Put** and **putb** actually require strings, and if there is just one string argument, **put** behaves like **print**.

Read and **reada** can read any value written by **write** or **writea**, except for atoms (produced by **newat**) and procedure values (produced by **routine**). **Reada** differs subtly from **getb** in that it absorbs all characters up to the end of the line after reading as many values as directed by the presence of **wr** arguments, whereas **getb** does not—the next **getb** will pick up where the previous one left off. This accords with the historical intent of these routines. (The **b** in **getb** and **putb** stood for "binary", a mode to which the notion of a line boundary was foreign.) Tokens representing input values need to be separated by commas and/or runs of whitespace.

Putfile is identical to **putc** except that **putfile** will automatically close an automatically opened file (see Section 2.10). **Puta** is the same as **putline**, and puts a line terminator (a newline character in Unix) after each string it writes. **Putc** does not so terminate lines.

The **getline** operator reads up to the end of the current line (or to the end of the file, if the line is not terminated), but does not return any line terminator.

All input functions and operators, if the end of file is encountered before any characters are read on the call, return **om**. Procedural forms such as **reada** and **geta** assign **om** to unsatisfied **wr** arguments.

2.3.2 String I/O

As in SETL2, it is possible to "read" from a string:

reads $(s, lhs_1, lhs_2, \ldots)$; -- $lhs_1, lhs_2, \ldots$ are **wr** args

This is not strictly compatible with the SETL2 version, which "consumes" value denotations from s and thus requires s to be writable as well as readable. The SETL version of **reads** $(s, lhs_1, lhs_2, \ldots)$ is roughly equivalent to

$[lhs_1, lhs_2, \ldots] :=$ **unstr** $(\text{'['} + s + \text{']'})$

except that **reads**, like **read** and **reada**, will tolerate trailing "junk" characters after a delimiter that terminates the last denotation needed to satisfy the last writable argument. See Section 2.14.2 for more information on **unstr**.

There is currently no corresponding **writes** or **prints** for string formatting. Arguably there should be, for completeness, but meanwhile, it is quite convenient simply to concatenate strings produced by the conversion primitives of Section 2.14.2 on those occasions when it is necessary to buffer intermediate string forms.

2.3.3 Direct-Access Files

For a stream opened in one of the direct-access modes listed at the beginning of this section, there are four special operations available, all of which employ the concept of a "current position" that is implicit in all file I/O. There is no distinction between input position and output position.

The current position can be explicitly set with the call

seek (*fd*, *offset*);

where an *offset* of 0 represents the beginning of the file. Also,

rewind (*fd*);

is equivalent to **seek** (*fd*, 0). Positions (offsets) are measured in characters.

For SETL2 compatibility,

```
gets (fd, start, length, lhs);          -- lhs is a wr arg
puts (fd, start, s);
```

combine position manipulation with I/O. Note that *start* obeys string indexing conventions, and must therefore be at least 1, corresponding to a file offset of 0. For **gets**, if the end of file is reached before *length* characters have been read, *lhs* will be assigned a string of fewer than *length* characters. **Gets** and **puts** update the current position after doing their reading or writing, respectively.

2.4 Signals and Timers

SETL programs can "read" high-priority signals from other processes, from the kernel, and from periodic timers. The I/O system is also used to specify signals that are to be ignored, meaning in most cases relieved of the duty of terminating the process.

Routing signals and timer alarms through the I/O system, and making their associated streams candidates for multiplexed event-sensing through the **select** routine described in Section 2.5, are of great value in helping to create small, modular processes that are simultaneously responsive to I/O events, signals, and the passage of time.

2.4.1 Signals

A SETL program arranges to receive signals of a given type by opening a stream on the signal name, such as INT, HUP, or TERM, contained case-insensitively in a string such as 'INT', 'HUP', or 'TERM' respectively, or even 'SIGINT', 'SIGHUP', or 'SIGTERM'.

The full list of signal names supported by a given Posix-compliant Unix system can be obtained from the kill command through its “list” parameter (a lowercase L):

```
kill -l
```

Descriptions of the signals can usually be found on Unix systems in the customary way, i.e., through a command like man signal, and further details are often available in the C “header” files, normally under the directory /usr/include. Section 2.8 shows how to send signals; the present section is about how to receive and process them.

To start intercepting signals, a SETL program executes

```
fd := open (signal_name, 'signal');        -- or 'signal-in'
```

For example, *signal_name* might contain ‘HUP’ or ‘SigHup’, in which case subsequent HUP signals sent to the SETL process will be caught and presented to the SETL program as lines of input on *fd*, one signal per line. When the SETL program detects that a signal of this type has been sent to it (this detection will often be through the **select** routine discussed in Section 2.5), it should explicitly receive the signal by reading a line from the stream’s file descriptor, e.g.,

```
line := getline fd;
```

At the time of this writing, the resulting *line* is specified only to contain at least the null string, but for some signal types, collateral information may eventually prove useful. To remain upwardly compatible, therefore, SETL programs should read whole lines from signal streams.

A signal type may have any number of streams open over it, and a line will be delivered to all of them whenever a signal of that type is caught.

If a particular type of signal is not being caught, because no streams are open over it, signals of that type may still be stripped of their default effect on the SETL process (which for many is to terminate the process) by being explicitly ignored:

```
fd := open (signal_name, 'ignore');          -- or 'ignore-signal'
```

The only meaningful thing that can be done with the *fd* returned in this case is to **close** it. When the last ignoring stream on the given signal type is closed, the default behavior of the signal type is restored unless there are by then signal-receiving streams open on that type. If there are both receiving and ignoring streams open on a given signal type, the receivers take precedence—incoming signals will be delivered rather than being ignored.

2.4.2 Timers

A SETL program may open any number of recurring interval timer streams based on "wall-clock" time, user-mode CPU time, or total CPU time:

```
fd := open (ms, 'real-ms');                          -- wall-clock time
fd := open (ms, 'virtual-ms');                   -- user-mode CPU time
fd := open (ms, 'profile-ms');                        -- total CPU time
```

The *ms* argument in each of these timer cases is actually a string consisting of decimal digits to be interpreted as the number of milliseconds that is supposed to elapse between each time a new line becomes available on that stream. These timer I/O modes make implicit use of the signals ALRM, VTALRM, and PROF respectively.

The file descriptors returned by **open** for signal and timer streams are “pseudo-fd’s” in that they have no existence at the Unix level. This decision to to route signals and timers through the SETL I/O system was made primarily so that **select** (Section 2.5) can sense timing and other signals simultaneously with regular I/O events.

2.5 Multiplexing with *Select*

Event-driven programs need to be able to wait for any of a set of I/O events simultaneously, and then identify which channels can be read or written without blocking the process.

In SETL, the routine to do this is called **select**, after the select routine introduced in 4.3BSD Unix. It takes a tuple of up to three sets of file descriptors as one parameter, and an optional timeout value as another. The sets identify streams that may become (1) ready for reading, (2) ready for writing, and (3) ready to return an error indication. The last of these has no specific meaning within SETL, though the environment may assign some. The most typical call, with only a set of potentially readable file descriptors specified, is

[*ready*] := **select** ([*readfds*], *timeout*);

and the general case is

$[r\_ready, w\_ready, e\_ready] :=$ **select** $([readfds, writefds, errorfds], timeout);$

The result set *ready* or *r_ready* is a subset of *readfds*, and lists those streams from which something can be read without blocking. Note that this does not say how many characters can be read, and in fact zero is possible, such as when end-of-file is immediate. Similar considerations apply to output, although in practice, operating systems and networks may themselves buffer packets and allow a program to flush all its output long before the receiver is actually ready.

Furthermore, *readfds* may include file descriptors on which **accept** (Section 3.1.2) can be called without blocking, and pseudo-fd's for signal and timer streams (Section 2.4.2) when they have lines to deliver.

The *timeout* parameter, which is an integer number of milliseconds, can be omitted to specify an indefinite wait, or can be as low as 0 to effect "polling".

We will find in Chapters 3 and 4 that it is convenient to structure virtually every TCP/IP server as a loop around a **select** call.

2.6 Files, Links, and Directories

Given a file descriptor, it is possible to recover the filename that was originally passed to **open** if that is available in the current process, as the string-valued expression

filename *fd*

As with the other I/O routines, *fd* can in fact be that original filename, in which case this call merely checks that the file is currently open and returns the filename. It is also possible to obtain the file descriptor corresponding to an open file as designated by its name or file descriptor, as the integer-valued expression

> **fileno** *fd*

Again, this is just a checking identity function for open file descriptors, sometimes used in the idiom

> $fd :=$ **fileno open** $(name, \ldots);$

as a way of ensuring that an **om** return from **open** immediately causes a run-time error.

To facilitate the use of SETL in a "shell" programming role without the need of resorting to running an external command, the boolean-valued expression

> **fexists** *s*

for any string *s* indicates whether a file named by the contents of *s* exists in the local environment. Similarly, the integer-valued expression

> **fsize** *s*

is the number of bytes in the file named in the string *s*, if the file exists.

Ordinary files are created automatically when they are first opened for writing, but the creation and manipulation of "links" requires the use of certain special functions. A "hard" link is created atomically by the following routine, if *existing* is a string naming a file that exists before the call, and *new* names a file that does not exist before the call:

link (*existing*, *new*); -- *existing* and *new* are filenames

After the call, *existing* and *new* are equivalent references to the same file. If *existing* does not exist before the call, or if *new* already exists, **last_error** is set to something other than **no_error** (see Section 2.13).

In a local filesystem, **link** can be used to implement a "test and set" mutex: assuming *existing* exists, then if *new* also exists, the operation will fail, but if it doesn't exist, then it will be created and the calling process will "own" the mutex lock until it releases it by calling **unlink** (see below) on *new*.

Similarly, a "symbolic" link can be created by the call

symlink (*s*, *new*); -- *s* is an arbitrary **string**

There is no requirement that a file named *s* exist beforehand in order for this call to succeed, although it will fail if *new* already exists. Thus **symlink** can be used to implement a mutex in much the same manner as **link**, but with the added benefit that *new* can be "pointed at" an arbitrary string. This may, for example, embed information about the process that currently holds the lock, a technique that is used by the vc-toplev.setl program listed in Section A.42.

In order to find out whether a particular symbolic link currently exists in the filesystem under a name given in a string *s*, the boolean-valued expression

lexists *s*

is used. Note that when **fexists** is applied to a symbolic link, it interrogates the existence of the file referred to by that link, whereas when **lexists** is applied to a symbolic link, it merely interrogates the existence of the link itself.

The use of **lexists** on a name intended to represent a mutex lock is unlikely to occur in code that is free of race conditions. For race-free operation, the following sequence has the requisite test-and-set atomicity:

```
clear_error;
symlink (my_id, lockfile);
if last_error = no_error then
  -- we have the lock
    ⋮
  unlink (lockfile);   -- release the lock (unlink is defined below)
else
  -- some other process has the lock
    ⋮
end if;
```

When *s* is known to name a symbolic link, its associated text is available as the string-valued

readlink *s*

If *s* names something that exists but is not a symbolic link, or something that does not exist, then **readlink** returns **om** and sets **last_error** according to which case applies. By contrast, an attempt to read data from *s* will fail if *s* names something nonexistent or a symbolic link pointing to something nonexistent (ultimately, since symbolic links can point to filesystem entries that are themselves symbolic links, up to some system-imposed limit on the number of indirection levels).

Finally,

unlink (*s*);

can be used to destroy a hard link or a symbolic link. When the last hard link to a file is destroyed in Unix filesystems, the file itself is destroyed as soon as the last process that has it open closes it. (Creating a file is the act which creates the first hard link to the file.) Thus **unlink** is the standard routine for destroying *any* file in Unix.

Sometimes, a program will desire the use of a "scratch" file, though this need is declining with the increase in virtual memory sizes. Because these will often have to reside in a shared public area, the primary consideration becomes simply that of choosing a unique filename. The Unix routine tmpnam is the scratch filesystem analogue of the SETL **newat** generator, and the nullary SETL primitive

tmpnam() -- trailing "()" optional, as usual

uses tmpnam to yield a string filename that is "reserved" for the calling program.

The current working directory, which is a notion supported by every modern operating system, is available in SETL as the string-valued

getwd -- trailing "()" optional

and can be changed using the call

chdir (*dirname*);

The **umask** routine pertaining to file access rights is explained in Section 2.7.

2.7 User and Group Identities

The routines in this set are useful only on systems that support the notion of users and groups thereof, and are most useful on systems that distinguish “real” from “effective” users and groups. On systems with no such support, the “getters” should be implemented to return 0, and the “setters” should act as no-ops:

```
uid := getuid();          -- get real user id
uid := geteuid();         -- get effective user id
gid := getgid();          -- get real group id
gid := getegid();         -- get effective group id
setuid (uid);             -- set at least the effective user id
setgid (gid);             -- set at least the effective group id
```

The **setuid** (**setgid**) procedure sets the real and effective user (group) id if the effective caller is the superuser. Otherwise, the effective id is set to the requested value if it matches the real id or it matches the id associated with the file to which `exec` was last applied prior to starting the SETL run-time and that file has the appropriate (“allow set-user-id” or “allow set-group-id”) bit set in order to permit just such a change of effective id. If the requested identity change cannot be performed, **last_error** (see Section 2.13) will show the error. If the call succeeds, however, the process acquires the privileges of the effective id. It is not a good idea to make the executable file of a public SETL interpreter allow setting of the user or group id like this, or any user will be able to execute arbitrary SETL programs using the user or group identity attached to the interpreter file, simply by inserting a **setuid** or **setgid** call.

An example of a superuser process that needs to be able to change “down” to an ordinary user process is a login daemon such as rlogind or a periodic process-spawning daemon such as crond. It is also sometimes convenient for one user to own a program which, when run by another user, has available the privileges of both, **setuid** being used to toggle the effective user id back and forth between the two users as necessary. For example, when my students at Lehigh University submit their programming assignments, they unwittingly run a SETL program which assumes the identity of the submitter (the real user id) when it picks up files from their private directories, and takes on my identity (the effective user id, the owner of the script from which the SETL program is launched) when it squirrels an image of the submitter’s files away in a private area allotted by me. No superuser privileges are required to implement such a scheme, though of course some care had to be exercised in designing those parts of the program which execute “as me”, so as to avoid such perils as allowing one student to consume all my disk space and thereby preventing other students from submitting their assignments.

The flags attached to a file depend on the filesystem hosting the file, and their exact interpretation depends on the host operating system. However, if the filesystem maintains, for each file, 3 ordered sets labelled User, Group, and Others, each set consisting of 3 bits labelled Read, Write, and Execute, then there is a Posix-based routine for controlling an environmental mask value which determines which bits are set when files are subsequently created by the SETL program:

```
mask := umask();            -- get file creation bit-mask
umask (mask);               -- set file creation bit-mask
```

The 9 bits represented by octal 666 are OR'ed with the inverse of this mask to determine what bits are initially set in new files. For example, a user who wants new files to be created with read and write access for the user (the owner), and read-only access for everyone else, might code:

```
umask (8#022#);              -- User {rw-}, Group {r--}, Others {r--}
```

This environmental file creation mask is inherited by processes, and every Unix shell in common use has a built-in umask command that calls the corresponding Unix umask function. On systems where this mechanism is absent, SETL implementations are expected to treat the **umask** setter as a no-op, and have the getter return 0.

The use of octal is such a natural and customary choice here that some programmers even tend to use it on the system command that normally manipulates these bits, chmod. For example, if it is desired to make a file foo universally executable after it has been created with the "octal 644" permissions suggested by the example above, this can be done with either of the following calls:

```
system ('chmod +x foo');     -- merge 8#111# with foo's current bits
system ('chmod 755 foo');    -- set foo's bits to {rwx}, {r-x}, {r-x}
```

The chmod command or subroutine can also be used to set the bits which cause the effective user or group id to be changed to that of the file owner (as opposed to the process owner) when the file is used to replace a process image by exec, thus giving the executing program the rights of the file owner together with the ability (via **setuid** or **setgid**) to switch between those and the rights of the process owner. For example, if you want other people to be able to update your database only through your foo command, you might execute:

system ('chmod u+s foo'); -- make foo allow **setuid** to thee or me

2.8 Processes and Process Groups

An open pump stream is always connected to a child process, whose process id (see Section 2.17.2) can be retrieved by the integer-valued expression

pid (*fd*)

If *fd* is omitted, **pid** just returns the process id of the calling process, like the Unix getpid routine.

The existence of a process can be interrogated by the expression

pexists *id*

which tests whether *id* is the process id of a process currently known to the host operating system. This will be **true** even if the referred-to process has already exited but not had its status "reaped" yet by some equivalent of **wait** (again, see Section 2.17.2).

Finally, to send a signal to any existing process, a SETL program can execute the built-in

kill (*id*, *signal_name*);

If *signal_name* is omitted, it defaults to 'TERM'. As with signal names passed to **open**, alphabetic case is not significant, and the prefix 'SIG' is optional. **Kill** also allows an integer-valued signal number to be used in place of a signal name. If *id* is positive, it is taken as a process id, but if it is less than –1, its absolute value is taken as a process group id (see the description of **getpgrp** below), and the signal is sent to all members of that process group. An *id* of 0 is equivalent to –**getpgrp**, i.e., all members of the caller's process group. An *id* of –1 causes the signal to be sent to all processes to which the caller has permission to send signals. For user-level (non-superuser) processes, –1 therefore stands for all processes owned by the user associated with the calling process (see **getuid** in Section 2.7).

Most signals cause process termination, sometimes with other side-effects such as creating a file core (an image of the final virtual memory state of the process). As described in Section 2.4.1, the recipient may also have altered its default signal-handling behavior. One signal, 'KILL', is designed to be impossible to trap in this way, providing a method of last resort to arrest otherwise uninterruptible processes.

A full discussion of the semantics of signals is outside the scope of this dissertation, but the Unix 98 documentation [154] on the sigaction function and on the <signal.h> C header file gives more details.

In Unix, each new process belongs initially to the process group of its parent. The value of

```
getpgrp          -- trailing “()” optional, as usual
```

is the process group id, an integer. The call

> **setpgrp**;

sets the process group id equal to the process id. If this is the first time the process has called **setpgrp**, it establishes a new process group, of which the calling process is the leader.

The main significance of process groups is in the distribution of signals. If the first argument to the **kill** routine described in Section 2.8 is less than –1, its absolute value is taken to be a process group id, and the signal passed to **kill** is sent to all processes in that process group. This can be convenient for taking down whole trees of processes at once, though in practice, more orderly shutdowns are often attempted first.

2.9 *Open* Compatibility

For backward compatibility, it is permitted in SETL to ignore the result of **open** (which in the old CIMS SETL returned **true** or **false**), and simply refer to open files by the same name as originally passed to **open** in all subsequent I/O calls, provided this name is unambiguous (which will *not* be the case when the same file is open more than once simultaneously, or when more than one child process identified by the same command string, such as ‘wish’, is open).

SETL2 programs, which must treat the “file handle” returned by the SETL2 **open** as an opaque entity, should be immediately portable to the SETL described here without change, except that some differences will be observed in the way values are presented by the corresponding versions of procedures like **printa**.

2.10 Automatically Opened Files

Files are *automatically* opened in 'r' or 'w' mode if a stream input or output operation is initiated on a previously unmentioned name. For an auto-opened input file, when an end-of-file condition is sensed, the stream is auto-closed. The only routine which auto-closes a stream that was auto-opened for *output* is **putfile**, making it a partner to **getfile** in using the environmental filesystem as an elementary persistent database.

In order to discover whether a particular stream is open (either as a result of automatic opening or a successful **open** call), the boolean-valued expression

is_open *f*

can be applied to a filename or file descriptor f without causing any side-effects such as the setting of **last_error**.

2.11 Opening Streams Over File Descriptors

In all the definitions in this chapter so far, the first argument to **open** has been a string giving a filename, command, signal type name, or number of milliseconds, but it can actually be an integer file descriptor, which may for example have been inherited, or produced by one of the low-level routines listed in Section 2.17.1. A major use of the 'rw' I/O mode (Section 2.3) is in fact to obtain a SETL stream over an inherited file descriptor about which the inheritor knows and cares nothing except that the stream is bidirectional. There is an example of this in the program lengths.setl in Section 3.2.2.

A pseudo-fd returned by **open** for a signal or timer stream cannot be used as a first argument to **open**. It would not be a very useful feature, since signal handlers do not survive the exec calls implicit in **exec** (Section 2.17.2), **system**, **filter**, and pipe or pump stream **open** (Section 2.2) calls.

2.12 Passing File Descriptors

The normal inheritance of file descriptors by child processes suffices for many purposes, one of the most typical being when servers use child processes to deal with clients, as is illustrated by several examples in Section 3.2.

However, there are cases where it is convenient to be able to pass file descriptors to existing processes. For example, an httpd daemon will often optimize performance by maintaining a pool of child processes. Rather than spawning a new process for each new request, the daemon may pass the file descriptor created for the corresponding connection down to an existing process for handling. Passing file descriptors "up" a tree of processes can also be useful, such as when a child process is appointed to perform an **open** and related activities that might result in a crash. If the child process completes this extended **open** operation successfully, it can safely pass the new file descriptor up to the parent process, whereas if it crashes, no harm is done. (Isolating delicate operations in small processes to limit the damage they can do is a running theme of this dissertation—cf. Chapter 3, and Sections 4.2.1, 4.2.2, 5.3.2–5.3.4, and 6.5.)

In order for a file descriptor to be passed, the sender and receiver must already share a pipe or pump stream for its conveyance. Given a writable file descriptor pd_1 for such an open stream, and a file descriptor fd_1 to be passed, the sender executes

send_fd (pd_1, fd_1);

and makes a rendezvous with the receiver, which has a readable file descriptor pd_2 for its end of the conveyance stream and executes

$fd_2 :=$ **recv_fd** pd_2;

Note that the resulting fd_2 is allocated in the receiver's process space, and may be numerically different from fd_1, even though it refers to the same kernel structure. It is an alias, chosen automatically in the same way as by the low-level **dup** routine described in Section 2.17.1.

The mechanism underlying this generalized passing of file descriptors is defined in Unix 98, but at the time of this writing is not yet present in all versions of Unix in common use. In some versions an alternative mechanism exists, though in some of those it is only available to superuser processes. This situation will surely improve, but programs which use **send_fd** or **recv_fd** cannot currently be considered fully Unix-portable in practical terms. Solaris is among the operating systems in which full fd-passing support is already implemented, however.

2.13 Normal and Abnormal Endings

Normally, I/O can proceed until it is time to call

> **close** (*fd*);

on a given file descriptor, *fd*. The most common reason for closing a stream is that an end-of-file condition has been encountered. The value of

> **eof** (*fd*)

is **true** or **false** depending on whether an attempt to read past the end of the input available from *fd* has been made. The *fd* parameter is optional on the **eof** call; the value of

> **eof**   -- trailing “()” optional

is **true** or **false** depending on whether the last input attempt on *any* stream failed on an end-of-file condition.

The responsibilities of **close** include flushing the output buffer if necessary, possibly waiting for a child process to complete, and releasing the buffer and kernel resources associated with the system-level file descriptor. It is also permitted to apply **close** to a file descriptor that is only open at the operating system level and not at the SETL level, as mentioned in Section 2.17.1.

For bidirectional streams, one direction may be closed without closing the other by calling

shutdown (*fd*, *how*);

where *how* is one of the predefined constants **shut_rd**, **shut_wr**, or **shut_rdwr**. Even if both directions are closed in this way, the file descriptor remains open at the SETL level. One use of **shutdown** is to cause an end-of-file condition to be signalled to a TCP peer (see Chapter 3), to indicate that the local process is done sending data but would still like to receive a reply. Stevens [194] mentions several others in conjunction with the Unix 98 shutdown routine (which **shutdown** also calls).

When **open** fails, it returns **om** instead of an integer file descriptor. Programs need not check for this possibility if a crash upon the first attempt to do I/O on **om** is acceptable behavior. When programs are run in subshells whose crashing does no harm to their environments, as is often the case when they are invoked through pump streams or by **system** or **filter**, such behavior may indeed be acceptable.

That possibility notwithstanding, he reason **open** tries to offer the caller a chance for recovery from errors instead of just crashing the process is that initiating access to external resources is something that can logically be expected to fail sometimes, whether because the system has run out of file descriptors or subprocesses available to the current process, or because a file was not found, many clients will want the chance to take specific recovery action.

Whatever the reason, the caller of **open** that chooses to check for **om** will find a rich variety of possibilities in what

last_error

can yield after an **om** return. Any SETL program that wishes to issue a detailed diagnostic for internally detected **open** failures does well to include the value of **last_error** in the error message, much as a C program obtains similar information from strerror.

Calling

clear_error;

will restore **last_error** to its default of returning

no_error

and the latest value of **last_error** will always depend on the latest setting of the Unix global errno variable by a system routine. For example, many of the low-level Unix (Posix) interface routines described in Section 2.17 will express failure by setting **last_error** rather than by abending the SETL process. It is a good idea to execute **clear_error** just before calling any routine that can interact with the external environment in any way, if one intends to inspect **last_error** (i.e., compare it to **no_error**) after the call.

On the other hand, particularly where networks are involved, either because of explicit use of sockets or because of networked filesystems, it is possible for practically any regular I/O operation to fail catastrophically, and one of the main reasons for delegating I/O responsibilities to child processes in the software designs preferred throughout this dissertation is to limit the damage caused by unpredictable communications failures—if the child running under a pump stream crashes, the parent simply sees an end-of-file condition on the pump stream.

2.14 Strings

Ultimately, all input and output reduces to the communication of strings. The importance of string handling in data processing languages was appreciated in both CIMS SETL [181] and SETL2 [190], which went beyond the already powerful string slicing operations and introduced a set of built-in procedures inspired by the intrinsic pattern-matching functions of SNOBOL.

2.14.1 Matching by Regular Expression

I have gone a step further and extended the string slicing operations themselves so that wherever an integer trimscript is required, a regular expression may be used instead. The regular expression is itself just a string in which certain characters called "metacharacters" are not meant to be taken literally, but act as patterns. The pattern-defining sublanguage is very similar to that accepted by the GNU egrep command. The predefined boolean variable

magic

may be assigned **false** to make all the metacharacters literal instead of special. Because **magic** is a global variable that defaults to **true**, the SETL programmer should normally set it back that way after any code sequence that requires it to be **false**, so for convenience there is also

old_magic := **set_magic** (*new_magic*);

where *old_magic* and *new_magic* are boolean. For example, a piece of code in a subroutine should set **magic** according to its local needs and then restore it:

```
saved_magic := set_magic (false);   -- we need metacharacters turned off
  ⋮                                 --  ... pattern-matching activity ...
set_magic (saved_magic);            -- restore prevailing value of magic
```

The string slicing extensions work as follows. Given string *s* and regular expression pattern *p*, the expression

$s(p)$

refers to the leftmost substring of *s* that satisfies *p* (and *p* itself will be "greedy" in what it matches wherever the Kleene star or other unbounded subpattern occurs). This expression may be used in store or fetch positions as usual, replacing or producing a substring accordingly. If there are no occurrences of *p* in *s*, then *s*(*p*) has the value **om**, and assigning to *s*(*p*) has no effect.

Given *s* and two regular expression patterns p_1 and p_2, the expression

$s(p_1 \,..\, p_2)$

refers to the substring of *s* which begins with the leftmost substring satisfying p_1 and ends with the first substring to the right of that satisfying p_2. For example, if *s* contains the text of a C program, the assignment

```
s('/\\*' .. '\\*/') := ' ';
```

replaces its first C comment (if any) with a blank.

We see here a consequence of the fact that the backslash is the “literal next character” indicator both in SETL strings and in the regular expression sublanguage. To match an actual asterisk, rather than have the asterisk in the pattern interpreted as a “0 or more occurrences” suffix operator (Kleene star), it is necessary to double the backslash. This produces a single backslash in the string value corresponding to the raw denotation, and this backslash in turn protects the asterisk in the regular expression.

Alternatively, of course, **magic** could be set to **false** so that

$s($‘`/*`’ `..` ‘`*/`’$) :=$ ‘ ’;

would have the desired effect.

Although I have found regular expressions for string slicing to be very useful, they do not provide an easy way to construct replacement strings as expressions in terms of matched substrings. This virtue is possessed by SNOBOL and by the standard editing tools in Unix, and is useful enough that I plan to add such a capability to SETL (see Section 6.4).

Meanwhile, the **mark**, **gmark**, **sub**, **gsub**, and **split** built-in routines for scanning and modifying strings help to cover much of the need for a more complete pattern-matching facility:

$[i, j] :=$ **mark** (s, p); -- $i, j :=$ integers such that $s(i \,..\, j) = s(p)$
$[[i_1, j_1], [i_2, j_2], \ldots] :=$ **gmark** (s, p); -- all occurrences of p in s
$x :=$ **sub** (s, p, r); -- $x := s(p)$; $s(p) := r$; [if no side-effects in p]
$x :=$ **sub** (s, p); -- $x := s(p)$; $s(p) :=$ ‘’; [if no side-effects in p]

$[x_1, x_2, \ldots] :=$ **gsub** (s, p, r); -- all (replaced) occurrences of p in s
$[x_1, x_2, \ldots] :=$ **gsub** (s, p); -- all (deleted) occurrences of p in s
$t :=$ **split** (s, p); -- $t :=$ tuple of p-delimited substrings of s
$t :=$ **split** (s); -- $t :=$ **split** $(s,$ '[\f\n\r\t\v]+'); [whitespace delim.]

Each pattern argument is denoted p in this synopsis. It may be either a regular expression as with the string slicing extensions, or an ordered pair $[p_1, p_2]$ of regular expressions, where p_1 and p_2 behave, in terms of matching, exactly like the p_1 and p_2 in the slicing form $s(p_1 \mathinner{..} p_2)$ just reviewed. As a matter of fact, p_1 and/or p_2 can be integers in all these forms, for full orthogonality in expressions like $s(p_1 \mathinner{..} p_2)$ or **mark**$(s, [p_1, p_2])$. **Gsub** returns the tuple of substrings of s that are replaced by r. **Gmark** does not rewrite s but returns a tuple of ordered pairs of (integer) indices such that every pair $[i_k, j_k]$ frames a substring of s that is entirely matched by the pattern p, or more precisely, such that $s(i_k \mathinner{..} j_k)(p) = s(i_k \mathinner{..} j_k)$.

More information on these and myriad other intrinsic operations comprising the SETL "library" can be found on the World Wide Web [19].

2.14.2 Formatting and Extracting Values

The next 3 routines, for formatting numbers in decimal, are named after functions in Algol 68 [137].

The string-valued expression

whole (i, width)

represents the integer *i* in decimal, with a possible leading minus sign. If the absolute value of the integer *width* is more than the number of characters in this converted number, then in the manner of printf, if *width* is positive, the number is right-justified in a field of *width* characters, and if negative, it is left-justified in a field of –*width* characters. If *i* is real, an integer nearest to *i* takes its place.

For a string that includes a possible decimal point and subsequent digits as well,

fixed (*x*, *width*, *prec*)

takes a real or integer *x*, a *width* that functions exactly as in **whole**, and a non-negative integer *prec* stating the number of digits to follow the decimal point. If *prec* is zero, **fixed** omits the decimal point as well, and in fact acts just like **whole** then.

For scientific notation, there is

floating (*x*, *width*, *prec*)

which differs from **fixed** only in that the character ‘E’ followed by a sign and at least 2 decimal digits are appended, representing the power of 10 by which the part before the ‘E’ is understood to be multiplied. That initial segment will have just one digit before the decimal point (if any). The **width** specification applies to the entire string.

Integers can also be rendered in explicit-radix form. The call

strad (*x*, *radix*)

for integers *x* and *radix*, given a value of *radix* in the range 2 to 36, produces a string of the form ‘*radix*#*digits*’, where the *radix* part is in decimal and the *digits* part consists of digits in the given radix. The convention is that the letters ‘a’ through ‘z’ are digits representing the values 11 through 36, respectively. Here are some examples:

> **strad** $(10, 10) =$ ‘10#10’
> **strad** $(10, 16) =$ ‘16#a’
> **strad** $(10, 2) =$ ‘2#1010’
> **strad** $(-899, 36) =$ ‘-36#oz’

The contents of the strings produced by **strad** would be acceptable as **integer** denotations if compiled as part of a SETL program, and would also be acceptable to the **read**, **reada**, and **reads** routines mentioned in Sections 2.3.1 and 2.3.2, as well as as to the **unstr**, **val**, and **denotype** operators described below. In all of these cases, another sharp sign (‘#’) may optionally be appended to the literal without changing its meaning.

Finally, the programmer can always use the general-purpose **str** operator to let the system choose how to format a given number. For integers, this will always be a decimal string, preceded by a minus sign (‘-’) if appropriate. **Str** also occurred in CIMS SETL and in SETL2.

As introduced in SETL2, the **unstr** operator is approximately the inverse of **str**. It cannot produce an atom (only **newat** can do that), nor a procedure value (only the **routine** operator can do that). Also, it is not guaranteed that $(1/3) =$ **unstr str** $(1/3)$, because there is no guarantee about how many digits **str** will produce. It is merely implementation advice that the number of significant digits yielded be close to, but not exceed, the precision of the machine representation of a SETL **real**, which should normally have at least 50 bits of mantissa.

When the programmer wishes to determine whether a given string *s* consists of a single valid numeric denotation (with possible leading and/or trailing whitespace), and obtain the corresponding value if it does,

val *s* -- this is **real** or **integer** or **om**

will yield the appropriate value. Note that the following identities hold for any integers *x* and *width*, and any *radix* in the range 2 to 36:

x = **val str** *x*
x = **val whole** (*x*, *width*)
x = **val strad** (*x*, *radix*)
x = **unstr strad** (*x*, *radix*)

By design, an important difference between **unstr** and **val** is that **val** is defined to return **om** when its argument *is* a string but *does not* consist of a numeric denotation, whereas the behavior of **unstr** is unspecified for invalid arguments. The intent is that SETL implementations raise some kind of exception when **unstr** cannot recognize a SETL denotation in its argument. At the time of this writing, there is no formally defined exception mechanism for SETL, though see Section 6.5. Meanwhile, checking implementations are expected to handle this kind of error in some manner helpful to programmers. For example, when my SETL implementation [19] detects an error at run time, it highlights a source line, points to a relevant token, and displays a subroutine traceback.

In order to determine whether a string would be acceptable to **unstr**,

denotype *s*

is defined as **type unstr** *s* if *s* consists of a valid SETL value denotation, but **om** otherwise. No exceptions!

2.14.3 Printable Strings

When **str** is confronted with a string argument, it increases the quoting level if necessary by surrounding the string with quote marks and doubling internal ones, but leaves all “unprintable” characters as they are. (The reason it may *not* be necessary to add quote marks is that the string may have the form of a SETL identifier—an alphabetic character followed by alphanumeric and underscore characters. **Str** and **unstr** are identity operators on strings with content restricted in exactly this way.)

The expression

pretty *s*

formats a string *s* such that all characters are represented as “printable” characters. Quotes and backslashes are doubled, the “control” characters are represented in C or SETL denotation form as shown in the following table, all other unprintable characters are rendered as a backslash followed by octal digits, and the remaining characters, all printable, are left as they are:

FORM	FUNCTION
‘`\a`’	audible alarm

‘`\b`’	backspace
‘`\f`’	formfeed
‘`\n`’	newline (linefeed)
‘`\r`’	carriage return
‘`\t`’	horizontal tab
‘`\v`’	vertical tab

The **pretty** operator also encloses the result string in quotes.

Conversely,

unpretty *p*

takes a pretty string *p* and performs the inverse operation. It is of course liberal enough even to accept some strings that **pretty** would not produce, though it does insist on the enclosing quotes (single or double).

Another operator which converts a string to another string having all characters printable is

hex *s*

which has the inverse

unhex *s*

so that **unhex hex** $s = s$. **Unhex** returns **om** if its argument fails to consist of an even number of hexadecimal characters, those being the decimal digits and the letters ‘`a`’ through ‘`f`’ in upper or lower case.

Hex is particularly useful for instrumenting low-level code in which special string encodings are used, such as when a serial-line device has a predefined command protocol. The Canon VC-C3 [35] videocamera system to which the control service described in Section 4.1.2 interfaces is a perfect example. Similarly, **unhex** makes it very easy to set up a low-level diagnostic tool, to allow the prober to throw arbitrary strings at the device. In programs such as vc-model.setl, listed in Section A.27, **unhex** can also be seen to serve the rather trivial but welcome purpose of facilitating the use of hexadecimal string denotations in the program text itself, thereby avoiding the need for '\x' escapes to be repeatedly embedded in string literals—a low-level aid to readability.

2.14.4 Case Conversions and Character Encodings

The expression

to_upper *s*

is the same as *s* except that all lowercase characters are converted to their corresponding uppercase forms, and

to_lower *s*

is the obvious complement. These case conversion operators are useful for canonicalizing a string such as might occur in an input command to a program, because then all subsequent tests or map lookups on the converted string can be effectively case-insensitive.

Following the CIMS version of SETL, the asterisk is overloaded to allow a string *s* to be "multiplied" by a non-negative integer *n* to produce the concatenation of *n* copies of *s*. The arguments can be in either order. For example, a row of 70 dashes can be specified as $(70 *$ '-'$)$ or $($'-' $* 70)$.

Likewise, **lpad**(s, n) and **rpad**(s, n) yield copies of *s* padded with blanks as necessary on the left or right, respectively, to make up *n* characters.

As in CIMS SETL, the **char** operator takes an integer that is the internal code of some character, and returns that character as a string of length 1. The **abs** operator is overloaded to act as **char**'s inverse, and

ichar *s*

is introduced in SETL as the equivalent of **abs** *s* when *s* is a string (and it is an error for *s* not to be a string).

The up-to-date reader will note that no distinction has been made between *bytes* and *characters* for SETL **string**s. In effect, only the "POSIX locale" defined in Unix 98 is accommodated by the current design of SETL, and characters are assumed to occupy 8 bits. However, the language is not strongly tied to this assumption, and can be expected to evolve gracefully toward support for "wider" characters and for contemporary internationalization and localization standards. Areas of the language for which compatibility issues will arise (although the new definitions should largely be upwardly compatible with the existing ones) include the **char** and **ichar** operators of this section, **hex** and **unhex**, escape sequences in **string** denotations, and direct-access I/O operations. Which characters are considered "printable", the collating order among strings, case conversions, the decimal point symbol, and the format of times and dates should all ultimately become locale-dependent. If the locale can be changed by a SETL program during its own execution, which does not seem unreasonable, there will also be dynamic convertibility concerns to be addressed.

2.14.5 Concatenation and a Note on Defaults

String concatenation is a very common operation, particularly when used for building up output strings. In principle, it is possible to require the SETL programmer to apply **str** to every value that is not already a string when building up a string, but in practice, it is much more convenient for the programmer if **str** is invoked implicitly. This is not by any means a context in which type mistakes are likely to be disastrous, and given that many strings are built for the sake of producing error messages, it is actually more likely that an important diagnostic prepared by the SETL programmer will be missed due to a gratuitous crash than that a critical type error will go unnoticed and have its deadly effects propagated far, if there is insistence on explicit coding of a **str** in front of every non-string expression in a long concatenation. For example, expressions that evaluate to **om** in this situation will show up as '$*$' in the concatenated string, and this itself conveys useful information.

It happens that the "+:=" operator is overloaded to support **om** as the initial value of its (writable) left-hand operand when the right-hand operand is an **integer**, **real**, **string**, **set**, or **tuple**, in which case it acts as if it had been initialized to the appropriate identity element (0, 0.0, '', {}, or [ ], respectively). This is helpful in loops, such as when tallies are being recorded against keys in a map, e.g.:

```
tally_map := {};
for x ... loop
```

```
  $tally\_map(x)$ +:= 1;
    ⋮
end loop;
```

Without the identity-element default, the statement "$tally\_map(x) +:= 1;$" above would have to be preceded by "$tally\_map(x)$?:= 0;", which in practice is a nuisance that is hard to justify by the need for protection against failure-to-initialize errors. But since the expression **om** + 'a' is supposed to be equivalent to **str om** + 'a' (which has the value '∗a') by the implicit-**str** rule, the question arises: should x +:= 'a' for uninitialized x mean x := 'a', or x := '∗a'? It is unusual to want to form a string starting with the converted value of **om** (indeed, the very use of "+:=" in a string-building expression that is itself meant to be copied somewhere is stylistically questionable), but it is not at all unusual to want automatic initialization of a string that will be accumulated by concatenation, so the decision is easily made in favor of the latter interpretation.

2.15 Field Selection Syntax for Maps

Pending the design and implementation of a type system for SETL, I have added the convenience of records to SETL (without, alas, the security of strong typing) by extending the member selection ("dot") notation so that it can be used to address range elements in maps whose domain elements are strings having the form of SETL identifiers. For any single-valued, string-domained map f, the identity

$$f.x = f(\text{'x'})$$

holds. The dot also retains its normal purpose of explicitly resolving member names to packages. In a case-insensitive stropping[3] regime, *F.X*, *F.x*, and *f.X* are also equivalent to $f(\text{‘x’})$ and $F(\text{‘x’})$ (but not $f(\text{‘X’})$, i.e. $F(\text{‘X’})$).

This small (and admittedly dubious) extension to the meaning of the infix dot, originally intended as a stand-in to allow maps to serve easily as records, proves to be very useful for program-to-program communication. With no preliminary declarations or difficulties about representing records of various types while they are in transit in the I/O system, SETL programs can pass data around as maps, where one field may serve as a discriminant (tag) to indicate which other fields are meaningful. Such flexibility offers more power than discipline, but at least corresponds to familiar practice in JavaScript [152] and resembles the model of "resources" in the X windowing system [121], so it can be argued that the pitfalls as well as the strengths of this approach are already fairly well known.

2.16 Time

As in CIMS SETL and SETL2, the current date and time are available as the string-valued **date**, or equivalently **date**().

Similarly, the integer-valued **time** in all versions and variants of SETL gives the

[3]"Stropping" was a term borrowed from barbers by the designers of Algol 68 [137] to describe how sequences of characters on an input device correspond to the abstract symbols (tokens) of a programming language. My SETL implementation [19] defaults to a regime similar to the RES ("reserved word") convention of Algol 68, and is general enough to support several other modes, including one like Algol 68's UPPER.

number of milliseconds of CPU time used so far by the current process and all child processes that have finished and been waited for by any equivalent of **wait** (see Section 2.17.2).

New in the present version of SETL are the integer-valued

clock -- trailing "()" optional

which measures the number of milliseconds of "wall-clock" time that have elapsed since the calling process started, and

tod -- trailing "()" optional

which is the number of milliseconds since the beginning of 1970 (UTC). Also new is

fdate (*ms*, *fmt*)

which formats its integer argument *ms*, presumed to represent a number of milliseconds since that moment, as a time and date according to the format given by the string *fmt*. The format is optional, and defaults to '%a %b %e %H:%M:%S.%s %Z %Y', where the per-cent sign "escapes" are as defined for the Unix 98 strftime function as applied to the result of applying its localtime function to *ms* **div** 1000, plus the extension that '%s' expands to the low-order 3 digits of *fmt* in decimal. For example,

fdate (936433255069) = 'Sat Sep 4 04:20:55.069 EDT 1999'

in the US Eastern time zone. The SETL **date** primitive, which formerly produced the time and date in an implementation-dependent format, is now standardized as meaning **fdate** (**tod**, '%c').

2.17 Low-Level System Interface

For most purposes, the high-level SETL model embodied in filters, pump streams, and so on, with its automatic buffering and process management, will be the most direct and convenient. Occasionally, however, access to certain of the Unix 98 [154] mechanisms underlying this model will be desired. The routines described in this section aim to be a supporting cast of utilities in this spirit, and have names that should be mnemonic to programmers familiar with Unix.

2.17.1 I/O and File Descriptors

The lowest-level Unix facilities for creating "pipes" and file descriptor aliases directly are now available in SETL. It should seldom, if ever, be necessary to use these primitives, but an example in Section 2.17.2 shows how these traditional tools could be used with **fork**, **exec**, and **wait** to implement piping from a child process in the manner of a SETL 'pipe-from' stream or a C stream obtained from popen in 'r' mode.

A Unix-level pipe is created with

[*rfd*, *wfd*] := **pipe**(); -- trailing "()" optional but recommended

which leaves a readable file descriptor in *rfd* and a writable file descriptor in *wfd*. These file descriptors are not open at the SETL level, but are open at the operating system level.

It is still easy to open a SETL stream over such a file descriptor, as shown in Section 2.11, and then all the appropriate SETL operations become available on it. Also,

close can always be called on any file descriptor, and will close all levels that are open. See Section 2.13 for more information on **close**.

To create a new file descriptor that refers to the same kernel object as an existing file descriptor, one of the following two calls may be used:

fd_2 := **dup** (fd_1); -- system picks fd_2
dup2 (fd_1, fd_2); -- we demand fd_2

In the first case, **dup**, the system chooses the lowest-numbered free file descriptor. In the second case, **dup2**, the caller chooses the desired file descriptor fd_2, and the system executes **close** on fd_2 if necessary before reopening it as an alias of fd_1.

For example, if a process wished to "redirect" the input from an inherited file descriptor fd_1 to its own **stdin** channel for convenience, it could execute

dup2 $(fd_1$, **stdin**$)$;

There is in fact a precise correspondence between **dup2** and the standard Unix 98 shell syntax for file descriptor redirection. For example, **dup2** $(6, 7)$ is equivalent to the shell's "7<&6" if file descriptor 6 is read-only, or "7>&6" if it is write-only.

Very occasionally, in contexts where such low-level operations are already being used, it is useful to bypass the SETL stream buffering and make direct calls to the following interfaces to the fundamental read and write Unix system primitives:

s := **sys_read** (fd, n);
n := **sys_write** (fd, s);

The **sys_read** procedure returns a string of up to *n* characters. **Sys_write** tries to write *s* to *fd* and returns an integer indicating how many characters of *s* it wrote. The call to **sys_write** should therefore normally be wrapped something like this:

```
procedure my_write (fd, s);
  while #s > 0 loop
    n := sys_write (fd, s);
    s(1 .. n) := '';
  end loop;
end my_write;
```

2.17.2 Processes

The fundamental process creation primitive in SETL is **fork**, which returns a new process id in the parent process and 0 in the child. The child is otherwise essentially a clone of the parent, and inherits its open file descriptors and signal and timer dispositions:

```
process_id := fork();          -- trailing "()" connotes action
```

See the vendor-specific fork Unix manual pages for more details. A feature of all fork implementations now in popular use is that they take advantage of the page modification flags supported by virtual memory hardware in order to defer actually copying pages of cloned data space until one process or another writes into them.

If there are insufficient resources available for the spawning of a child process, **fork** returns **om**.

Direct use of **fork** is rarely necessary, because one of **system**, **filter**, **pump**, or an I/O mode that starts a child process (see Sections 2.2.2 and 2.2.3) will provide a more convenient fit to most data processing needs. Still, it is helpful to have some appreciation of the Unix process model upon which the SETL model is built.

If a process which calls **fork** is still active when the child completes, it should clear the system's record of the child's process id and exit status by calling **wait**:

id := **wait** (*block*); -- *block* is **boolean**

If the flag *block* is **false**, **wait** will return immediately with either the process id of a child process that has exited, or 0 if none has yet. If *block* is **true** (the default), **wait** earns its name by waiting until a child process does exit, and then returns its process id. An important side-effect of a **wait** call that succeeds in "reaping" a process id in this way is that the record of the child process is cleared from the kernel's process table.

The high-level SETL operations that invoke child processes all effectively call **wait** at the appropriate time, namely when **close** is called on a pipe or pump stream (normally by the SETL programmer, but otherwise upon program exit), or during the last stage of a **system** or **filter** call.

After a successful **wait**, implicit or explicit, the exit status of the child process is available as the integer-valued

status

which is 0 if no child has yet exited. Note that a SETL program can exit with a particular status using the optional integer argument of the **stop** statement, e.g.:

stop 12; -- a multiple of 4 for IBM 360 nostalgia :-)

In the Unix realm, one of the first things a child process that is working at the level of **fork** will usually do, after some file descriptor rearrangement using **dup2**, is to replace its entire memory image with a new program, using **exec**:

exec (*pathname*, *argv*, *envt*);

The *envt* parameter is optional, and if it is absent, the *argv* parameter is also optional. The *pathname* is a string, and the optional parameters are tuples of strings if supplied. By default, *argv* will be taken to be [*pathname*].

Exec does not return, but constructs a call to the Unix routine execve if *envt* was supplied, or execv if it was not. See the manual pages for those routines for more information on how the pathname is used to find the executable file, and how the argument lists (*argv* and *envt* in SETL, mapped in the obvious way to null-terminated arrays of NUL-terminated C strings) are seen by the program when it is newly launched.

Here is a sketch of how **fork**, **pipe**, **dup2**, **exec**, **wait**, and **close** can be used in classical Unix style to provide a near-equivalent of the SETL 'pipe-from' input mode:

```
[rfd, wfd] := pipe();          -- create pipe
process_id := fork();
if process_id = 0 then
  -- Child, redirects stdout into pipe
  close (rfd);                 -- close readable side of pipe
  if wfd ≠ stdout then
    dup2 (wfd, stdout);        -- redirect stdout
    close (wfd);               -- now retire this alias
  end if;
  -- String name identifies an executable file:
  exec (name, args ...);       -- replace process image
  assert false;                -- exec should not return
```

```
end if;
-- Parent or only process continues here
close (wfd);                    -- close writable side of pipe
if process_id ≠ om then
  ⋮                             -- read from child via rfd, until EOF
  wait;                         -- clear child record from kernel
else
  -- No child was spawned
  printa (stderr, 'fork() failed');
end if;
close (rfd);                    -- done with readable side of pipe
```

See also the **pid** (process id), **pexists** (process existence), and **kill** (send signal) operations described in Section 2.8.

2.18 Summary

This chapter has covered the main needs one encounters in typical data processing, and reviewed several new features which help SETL participate meaningfully in a wide variety of process arrangements. Next, we turn our attention to more specifically Internet-related matters.

Chapter 3

Internet Sockets

Life on the Internet revolves around processes called *servers*. Accordingly, this chapter introduces the facilities now built into SETL for creating, communicating with, and governing servers. SETL proves to be well suited for expressing servers and their corresponding *clients*, and indeed the main goal of this dissertation is to show how convenient it has become for SETL servers to manage fluctuating sets of clients in data processing over the Internet.

The client-server conceptual model has been a huge success, to the point where for any pair of processes communicating over the Internet, it is helpful to label one as client and the other as server. Of course, we are speaking of relationships here, so a server can play client to other servers. In SETL, these relative roles are reflected in the names of the **open** I/O modes that create Internet sockets.

The usual job of a server is to wait for client requests and respond to them in some way. In the spirit of using processes as the fundamental modules of a data processing system, a server will typically define an interface consisting of some set of commands (or methods, in object-oriented terms) that is independent of the host operating system, hardware, and source programming language.

A server is in an ideal position to synchronize access to a resource, and will often be a long-lived process that consumes no CPU time and little memory while passively waiting for clients.

In order to remain available and responsive to client requests at all times, and to remain immune to client-induced crashes, a server will usually deal with each client through a pump stream (see Section 2.2.3). If the child process associated with that stream goes down due to a network failure or unhandled data exception, the server then merely sees an end-of-file condition on the stream.

3.1 Clients and Servers

Clients identify servers through a 'host:port' Internet naming convention, where the host part can be a DNS-recognized name or an IP address (see Section 3.3.2). Servers only bind the port part explicitly, since the identity of the host providing the service is implicit. If the server host is multi-homed (i.e., has more than one IP address), clients can reach that service through any address to which they have a route. The I/O modes specified to **open** for the creation of client and server sockets are distinct, as shown in the following list:

MODE AND SYNONYMS	MEANING
‘socket’, ‘client-socket’, ‘tcp-client-socket’	TCP client socket
‘server-socket’, ‘tcp-server-socket’	TCP server socket
‘udp-client-socket’	UDP client socket
‘udp-server-socket’	UDP server socket

This list of network-oriented I/O modes completes the list begun in Sections 2.2.2 and 2.3. Again, the mode parameter to **open** is *not* case-sensitive.

3.1.1 A Client

To open a client socket connected to a TCP server on an Internet host at a particular port, specified as a string *host_and_port* in which the host name is followed by a colon and then the port number, the SETL program executes

```
fd := open (host_and_port, ‘socket’);          -- or ‘client-socket’
```

For example, here is a complete SETL program to open a TCP client connection to port 13 (the “daytime” service) of host galt.cs.nyu.edu, read a line from the resulting stream, and print it:

```
print (getline open (‘galt.cs.nyu.edu:13’, ‘socket’));
```

This should produce that host’s impression of the current weekday, date, and time, e.g.,

```
Thu Jul 16 21:31:52 1998
```

For such “well known” port numbers as 13, which are generally listed with their familiar names in the file /etc/services on Unix hosts, the port number can be replaced by a service name such as daytime, allowing the above program to be written as

```
print (getline open (‘galt.cs.nyu.edu:daytime’, ‘socket’));
```

Running either of the above programs is approximately equivalent to issuing the following command on Unix or any system that has a telnet command:

```
telnet galt.cs.nyu.edu 13
```

As we shall see, telnet can be very useful for testing servers.

The client above is a little unusual in that the file descriptor returned by **open** is not even saved anywhere, since it is only used once (by **getline**). The following is more typical, and illustrates error checking comparable to what might be done after attempting to open a file:

```
fd := open (‘galt.cs.nyu.edu:13’, ‘client-socket’);
if fd ≠ om then
  print (getline fd);
  close (fd);          -- redundant (automatic on exit)
else
  print (command_name + ‘: ’, last_error);
end if;
```

This program should print either the response from the server or something like

```
<program-name>:  Connection refused
```

where **<program-name>** is the name of the SETL run-time interpreter or the name of the file containing the above SETL "script" if it is prefaced by a "#!" line and made executable as described in Section 2.1.

3.1.2 A Server

Whereas a TCP client simply has to request a bidirectional stream connection and wait for it to be established, a TCP server has to be able to perform two quite distinct steps in providing a service. First, it must be able to *listen* for connection requests from anywhere, and second, it must be able to *accept* such requests, establishing a distinct bidirectional stream connection for each accepted client.

As in the Unix 98 C interface, and more recently the Java API, one file descriptor is used for listening, and another is used for each client-specific connection produced by the **accept** routine discussed below.

The listening file descriptor is the one created by **open** when the 'server-socket' I/O mode is selected. The first argument is a string consisting of decimal digits identifying a port number on which to listen:

```
server_fd := open (port_number, 'server-socket');          -- "listen"
```

If the result of this call is not **om**, a listening TCP port has been created. To use the file descriptor associated with that listening port, the following call is used to wait for a new connection request from a client. When one arrives, this routine will unblock and yield a new bidirectional socket stream:

```
    fd := accept (server_fd);                          -- accept client connection
```

If another connection request arrives in the interval when the server is busy after a successful **accept** and before it has managed to call **accept** again, that request is queued. It will be immediately satisfied when **accept** eventually is called. Up to five connection requests will be so queued; beyond that they will be refused. Most servers are structured so as to spend very little time dealing directly with clients, thus keeping this queue short. Often, this means spawning a child process to handle most of the interaction.

For our first example of a server, however, let us consider a “sequential” server which innocently trusts clients to read what it sends to them immediately rather than delaying and thereby blocking service to other potential clients. With appropriate authentication, this kind of simple arrangement can be useful for resource access serialization. Here, the service is just supposed to mimic the “daytime” service of which the opening example of Section 3.1.1 was a client, except that this one listens on port 50013 rather than port 13:

```
server_fd := open (‘50013’, ‘server-socket’);                   -- listen
loop                                                -- cycle indefinitely
  fd := accept (server_fd);                    -- accept client connection
  printa (fd, date);               -- send current date and time to client
  close (fd);                                   -- close client connection
end loop;
```

There are two file descriptors involved here: *server_fd* is the one on which **accept** waits for client connections, and *fd* represents the TCP stream created by **accept** when a client does connect. It is not possible to send or receive data on *server_fd*: its one and only purpose is to listen for new client connections through **accept**. Conversely, **accept** does not create a listening server socket—that is **open**'s job—the only purpose of **accept** is to create a new socket representing the server side of an individual connection after a client requests one, and return a file descriptor for the bidirectional stream embodied in the socket.

Once a TCP connection is established between a client and a server, the relationship between the two parties is essentially symmetric, and in the standard terminology, each one is called the *peer* of the other. From the SETL programmer's point of view, the symmetry is in fact complete enough that when a child process inherits from a server process a socket file descriptor that is connected to a client, it can open a SETL stream over it using the **open** mode 'socket' (even though this nominally refers to a TCP *client* socket stream) instead of the more generic mode 'rw' as suggested in Section 2.11. In doing so, the child process exchanges some generality for the ability to make socket-specific enquiries on the stream such as those listed in Section 3.3.2.

3.1.3 Choosing the Port Number

The above server should behave similarly to the standard one on port 13 of most Unix and VMS systems. Port numbers in the range 1–1023, however, can only be served by processes with superuser privileges in Unix, and are what the *Internet Assigned Numbers Authority* (IANA) [116] calls the *well-known* ports. The best known of these will always be listed in the file /etc/services on Unix systems, together with their associated names such as daytime for port 13. Ports 1024 through 49151 are the IANA *registered* ports, which simply means that the IANA registers and maintains a public list of them at ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers. Finally, ports 49152 through 65565 (the maximum possible port number) are called the *dynamic and/or private* ports by the IANA. These have no preassigned association, and are conventionally also called *ephemeral* port numbers.

Historically, many Unix systems, particularly those with a BSD heritage, have allocated port numbers dynamically from the range 1024–5000, so these will often also be included in the “ephemeral” ports on a Unix system. Solaris systems allocate ephemeral ports from the range 32768–65535. Indeed, there is no guarantee in general that a port number will be available at the time it is requested by a would-be server process, since the port may already be in use. This is true even for IANA registered port numbers.

New server software should strive to avoid depending on a specific port number, especially if it is user-level software that is not ineluctably tied to a well-known port. Fortunately, this is easily done by requesting port 0, which instructs the system to choose an ephemeral port number. The assigned number can be found out using the **port** operator. For example, the value of

port *server_fd*

immediately after a successful **open** in the server of Section 3.1.2 would be the integer 50013 because ‘50013’ was the first argument of **open**. The following sequence of

SETL statements will print a number in the range 1024–4999 on a Berkeley-derived TCP implementation, or in the range 32768–65535 under Solaris:

```
server_fd := open ('0', 'server-socket');
print (port server_fd);
```

A client socket implicitly uses an ephemeral port number for its own end of a connection. This also can be interrogated with the **port** operator, though there is rarely any reason to do so. By contrast, a server may sometimes wish to know the ephemeral port number associated with a client on the client's host, and the **peer_port** operator described in Section 3.3.2 allows it to do so.

The primordial question naturally arises as to how a client can know what port a desired service is currently being offered on, if the port number was arbitrarily chosen only after the server program began execution. One way of handling this is to have the server make the port number known to a Web server (httpd daemon), and have clients initially contact the Web server to find out the port number of the desired service—Web servers listen on the well-known port 80 or are configured to use some other fixed port number. This two-stage technique is used in the case study of Chapter 4, where a Web document template is instantiated with dynamically assigned server port numbers and other information in response to initial client requests. In this chapter, however, for the sake of simplicity in server examples intended to illustrate other points, fixed port numbers are used.

3.2 Concurrent Servers

Except when the purpose of a server is to serialize access to some resource, it should be available to clients at all times. For example, a public Web server ought to be able to manage several connections simultaneously, and hand off most of the responsibility for clients to some equivalent of separate threads or processes so as to be able to accept new clients quickly.

In this section, a running example called the *line-length server* is introduced. Its actual function is quite trivial, in order not to obscure the issues confronting a server involved in extended interactions with multiple clients. For each line of text it reads from a client, the line-length server simply replies with a number indicating how many characters long that input line was. The number is itself formatted as a run of decimal digit characters on a single line.

3.2.1 Naïve Server

The first version of the line-length server does not even check for errors:

```
-- Line-length server, version 1 (naïve)

server_fd := open ('54001', 'server-socket');                -- listen
loop                                                         -- cycle indefinitely
  fd := accept (server_fd);                                  -- accept client connection
  if fork() = 0 then
    -- Child process; deal with client through fd
    while (line := getline fd) ≠ om loop
      printa (fd, #line);                                    -- number of chars in line
    end loop;
```

```
      stop;                                 -- normal exit from child process
    end if;
    -- Parent continues here
    close (fd);          -- child copy of fd stays open as long as necessary
  end loop;
```

This server will indeed serve any number of clients simultaneously, subject only to system resource availability, but it has some problems.

First, there is a subtle consequence of the fact that the file descriptor ostensibly returned by **accept** is not checked for being **om**. On rare occasions, **accept** will unblock because of an incoming connection request, and then, due to any number of network hazards, fail to establish the TCP connection. The child process will crash on its first attempt to use **om** as a file descriptor, which is unfortunate but does not affect the parent. However, the parent itself will crash when it calls **close** on **om**, rendering the whole service unavailable. We will solve this problem simply by conditioning everything after the **accept** call on *fd* not being **om**.

The second problem with this first version of the line-length server is that on Unix systems, where child processes can return a status code to their parents, the operating system is required to keep a record of the status code until the parent asks for it using wait or one of its variants. If we start the server as shown, have some clients use it and close their connections, and then ask ps to tell us about our processes, it will list all child processes that have finished interacting with clients (and hence exited) as *zombies*, the technical term for processes that have terminated but not yet had their status codes “reaped” by wait. Eventually, the system will not be able to allocate any more child processes, **fork** will start consistently returning **om**, and all further clients will be dropped immediately after they have been accepted. We will see in Section 3.2.3 how to be informed of when to call **wait**, the SETL reflection of wait that was described in Section 2.17.2, in order to clear zombies from the process table (which is of finite size).

The problem underlying the need for this compulsory housekeeping of calling **wait** is that **fork** itself is an unnecessarily low-level function. **Fork** was listed among the Posix interface routines in Section 2.17.2, and is really intended for system-level, not application-level work. There is almost always a better way of starting child processes in SETL using higher-level facilities such as **system**, **filter**, or a pump stream as previously described.

3.2.2 Shell-Aided Server

For example, we can fix the above problems by checking *fd* and using **system** to start child processes in the "background":

```
-- Line-length server, version 2 (shelly)

server_fd := open ('54002', 'server-socket');                 -- listen
loop                                                  -- cycle indefinitely
  fd := accept (server_fd);                      -- accept client connection
  if fd ≠ om then
    -- Convert fd to string, form command with it, run in background
    system ('setl lengths.setl -- ' + str fd + ' &');
    close (fd);          -- this has been inherited by the background task
  end if;
end loop;
```

The program lengths.setl that is run in the background for each client is as follows:

```
-- "lengths.setl"

-- Convert command-line parameter to integer, open r/w stream over it
fd := open (val command_line(1), 'rw');
while (line := getline fd) ≠ om loop
  printa (fd, #line);               -- line length
end loop;
```

The file descriptor is inherited by this child program, and identified on the command line that starts the child. In the child, the string token is converted using **val**, and the resulting integer is opened as a bidirectional SETL stream using the 'rw' mode. The trailing ampersand on the command invocation in the server is standard shell syntax to indicate that the shell should run the command in the background, i.e., as an independent, concurrent process that does not automatically receive keyboard-generated signals even if its (foreground) parent does.

Because **system** uses the shell to run commands, the executing instances of the child program in the above example are not direct children of the server, but rather of the shell, which exits (returns to the caller of **system**) immediately after launching the background process. In Unix, such "orphaned" processes automatically become children of the permanently resident init process, which then also takes over responsibility for reaping their status codes and thereby clearing them from the operating system's process table when they terminate.

3.2.3 Shell-Independent Server

If we wanted to avoid the use of the shell, perhaps on the grounds of a weak performance argument, syntactic allergy, or dependency paranoia, and happened to be familiar with the Posix API, we could code the line-length server in much the same way as it would be done in C or Perl, in contempt of the high-level approach. The following version does just that, and adds logging on **stderr** as a feature:

```
-- Line-length server, version 3 (posixy)

const server_port = '54003';

server_fd := open (server_port, 'server-socket');                  -- listen
if server_fd = om then
  -- Cannot get server port
  printa (stderr, 'Port', server_port, '-', last_error);
  stop 1;                                          -- exit with status code = 1
end if;

-- Arrange to receive CHLD (child exit) signals
sigchld_fd := open ('CHLD', 'signal');

loop                                                     -- cycle indefinitely

  -- Wait for listener and/or signal input
  [ready] := select ([{server_fd, sigchld_fd}]);

  if server_fd in ready then                       -- client wants to connect
    fd := accept (server_fd);                           -- accept connection
    if fd ≠ om then                                                 -- got it
      child_pid := fork();
      if child_pid = 0 then                                  -- child process
        -- Deal with client through fd
        while (line := getline fd) ≠ om loop
          printa (fd, #line);                                  -- line length
```

```
         end loop;
         stop;                                    -- exit with status code = 0
        end if;
        -- Parent or only process continues here
        close (fd);                                             -- child uses fd
        printa (stderr, child_pid, 'started at', date);
      end if;
    end if;

    if sigchld_fd in ready then                    -- a child process has exited
      line := getline sigchld_fd;                               -- take the signal
      child_pid := wait();                   -- get child process id and exit status
      printa (stderr, child_pid, 'rc =', status, 'at', date);
    end if;

  end loop;
```

Here we are also checking *server_fd* for **om**. This was not necessary in previous versions of the server, because the server would have crashed in an immediate and obvious way when the **om** value was passed to **accept**. Here, however, the **om** would enter silently into the set passed to **select** (the I/O event-waiting routine introduced in Section 2.5), and the program would simply sleep indefinitely, waiting for a CHLD signal through the remaining singleton set containing just *sigchld_fd*.

3.2.4 Pump-Aided Server

If more elaborate communication between parent and child were desired, such as a reporting of the number of lines and characters served, the hardy Posix API enthusiast might even go so far as to code the appropriate **pipe**, **dup2**, and **close** calls. But it is much easier to let the **pump** primitive take care of all such low-level housekeeping.

In the following version of the line-length server, a set is used to keep track of all the pump file descriptors. There is no need to catch CHLD signals any more, because child termination is reflected as an end-of-file condition on the child's pump stream, and the compulsory **wait** is implicit in the **close** then applied to that stream's file descriptor:

```
-- Line-length server, version 4 (pumpy)

const server_port = '54004';

server_fd := open (server_port, 'server-socket');                    -- listen
if server_fd = om then
  -- Cannot get server port
  printa (stderr, 'Port', server_port, '-', last_error);
  stop 1;                                         -- exit with status code = 1
end if;

pumps := {};                                   -- pump stream file descriptors

loop                                                     -- cycle indefinitely

  -- Wait for listener and/or pump stream input
  [ready] := select ([{server_fd} + pumps]);

  if server_fd in ready then                         -- client wants to connect
    fd := accept (server_fd);                             -- accept connection
    if fd ≠ om then                                                  -- got it
      pump_fd := pump();
      if pump_fd = −1 then                                    -- child process
        -- Deal with client through fd
        dup2 (stdout, stderr);                -- like shell 2>&1 redirection
        lines := 0;
        chars := 0;
        while (line := getline fd) ≠ om loop
          printa (fd, #line);                                   -- line length
          lines +:= 1;
          chars +:= #line;
        end loop;
        printa (stderr, 'lines =', lines, 'chars =', chars);
        stop;                                       -- exit with status code 0
      end if;

      -- Parent or only process continues here
      close (fd);                                   -- child (if any) uses fd
      if pump_fd ≠ om then                            -- child was created
        printa (stderr, pid (pump_fd), 'started at', date);
        pumps with:= pump_fd;                  -- include pump_fd in pumps
      else                                          -- no child was created
        printa (stderr, 'pump() failed at', date);
      end if;
    end if;
  end if;

  for pump_fd in ready * pumps loop             -- for each child with output
    child_pid := pid (pump_fd);                         -- process id of child
    child_output := getfile pump_fd;                  -- child's entire output
    close (pump_fd);          -- close the pump stream and clear the zombie
    printa (stderr, child_pid, ':', pretty child_output,
                    'rc =', status, 'at', date);
    pumps less:= pump_fd;                      -- remove pump_fd from pumps
  end loop;

end loop;
```

The purpose of **pretty** here is to ensure that the child's output is a legible part of the final **printa** statement about that child. For normal output, this will just be the report of lines and characters, as a quoted string. If the child crashes, as could happen if the client closes the connection without reading the reply to the last line it sends the server, this "pretty" string will contain any diagnostic output that might appear on either **stdout** or, because of the **dup2** call (which creates aliases as described in Section 2.17.1), **stderr**.

3.3 Defensive Servers

The server in Section 3.2.4 protects itself quite well against careless or malicious clients by handing them off to a child process immediately after they are accepted, but does not guard against the buildup of clients that somehow never get around to closing their connections. An example of a service which does time out in this way after 15 minutes of idle client time is FTP. If it did not do this, a popular FTP server would soon swamp its own host with idle TCP connections to its command port, since users naturally tend to leave such tedious housekeeping details as closing connections to the software rather than disconnecting explicitly.

3.3.1 Time-Monitoring Server

For a concurrent line-length server, there is not only the TCP connection but also the child process that consumes space on the server's host. What we want to do, if there is no client activity for, say, 15 minutes, is drop the client, forcibly if necessary.

In the following version of the line-length server, like the shell-aided server of Section 3.2.2, the overall parent process does nothing more than instantiate some external program for each new client, using the shell for convenience:

```
-- Line-length server, version 5 (impatient)

server_fd := open ('54005', 'server-socket');                   -- listen
loop                                                  -- cycle indefinitely
  if (fd := accept (server_fd)) ≠ om then                   -- new client
    -- Form command using fd converted to string, run in background:
    system ('setl impatient.setl -- ' + str fd + ' &');
    close (fd);                          -- this has been inherited by the child
  end if;
end loop;
```

Here, the program impatient.setl replaces the program lengths.setl of Section 3.2.2, but as we shall now see, prevents clients from "hanging" it indefinitely. If clients could be trusted to send and receive entire lines, it would be a simple matter of using **select** with a timeout argument of 15 minutes, like this:

```
  -- "impatient.setl" (too-trusting version)

  fd := open (val command_line(1), 'rw');          -- open inherited fd as fd
  loop                                     -- cycle until EOF on fd, or timeout
    [ready] := select ([{fd}], 15 * 60 * 1000);              -- 15-minute limit
    if fd in ready then                                       -- fd input or EOF
      if (line := getline fd) ≠ om then
        printa (fd, #line);                         -- send length of input line
      else
        stop;                                          -- exit on client EOF
      end if;
    else                                -- timeout (nothing from fd in 15 minutes)
      stop;                                        -- exit on client timeout
    end if;
  end loop;
```

But a devious client could send part of a line, and then the **getline** call would block indefinitely.

The solution to this problem, shown below, has impatient.setl fork itself into (1) a worker process which deals with the client, and (2) a monitor process which kills the first process if 15 minutes of continuous client inactivity occurs. The way this works is that the worker process sends the monitor process an empty line after each cycle of interaction with the client. If the monitor does not receive such a line within 15 minutes initially or after the previous cycle, it sends the worker a termination signal:

```
-- "impatient.setl" (highly cautious version)

if (pump_fd := pump()) = -1 then
  -- Child (worker), deals with inherited client stream
  fd := open (val command_line(1), 'rw');                -- inherited stream
  while (line := getline fd) ≠ om loop
    printa (fd, #line);                        -- send line length to client
    print; flush(stdout);                   -- make monitoring parent happy
  end loop;
  stop;                                                          -- all done

elseif pump_fd ≠ om then
  -- Parent (monitor) continues here
  loop                          -- cycle until EOF on pump_fd or timeout
    [ready] := select ([{pump_fd}], 15 * 60 * 1000);
    if pump_fd in ready then
      if getline pump_fd = om then
        stop;                          -- child exited normally, and so do we
      end if;
    else                        -- timeout (nothing from pump_fd in 15 minutes)
      kill (pid (pump_fd));                       -- send TERM signal to child
      stop;                           -- presume child exited, and do likewise
    end if;
  end loop;

else
  -- No child was created
  printa (stderr, 'pump() failed, dropping client');

end if;
```

If the monitoring parent process here had responsibilities other than closing *pump_fd*, the **stop** statements would become **quit loop** statements, and **close** would be called explicitly on the pump file descriptor. But to avoid clutter, the closing is left to be done automatically in this version, and the logging of information (*cf.* the server in Section 3.2.4) is omitted.

3.3.2 Identity-Sensitive Server

For security reasons, it is often important to know the identity of clients, so the following primitives have been introduced into SETL:

```
name    := peer_name fd;
address := peer_address fd;
portnum := peer_port fd;
```

Both *name* and *address* are returned as strings; the only difference between them is that *address* is the customary external representation of an IP address (4 decimal fields in the range 0–255, beginning with the high-order part of the address, and having the fields separated by dots), and *name* is an Internet primary host name if one can be found for the peer connected to *fd*, otherwise **om**. *Portnum* is the integer-valued port number of the peer connected to *fd*. Although the argument in all three of these cases is shown as *fd* to suggest an integer-valued file descriptor, this can as usual be the argument originally passed to **open** if that is known to the current process. These functions are primarily intended for servers to obtain information about clients, but are also available for client sockets, where they return information about servers. In the case of **peer_port**, of course, this is merely the number that was originally after the colon in the original argument to **open**, whereas when **peer_port** is used by a server to inquire about a client, the ephemeral port number it returns can be a useful way to distinguish among multiple clients from a single host.

The primary name and IP address of the host on which the current process is running can be obtained through the following string-valued nullary primitives:

```
name    := hostname;
address := hostaddr;
```

Finally, since a single host can have more than one name and/or IP address, the following primitives return sets of strings:

```
names     := ip_names (name_or_address);
addresses := ip_addresses (name_or_address);
```

The *name_or_address* argument to both of these is optional, and is understood to be that of the local host if omitted.

Just as a single host name can map to multiple IP addresses when the host has multiple network interfaces, a single IP address can almost always be reached by both a “local” name and a “fully qualified” name, and often also some further aliases. For example, consider this program:

```
print (ip_names ('birch'));
print (ip_addresses ('birch'));
print (ip_names ('128.180.98.153'));
print (ip_names ('genie'));
print (ip_addresses ('genie'));
```

It produces the following output when executed on host birch:

```
{birch 'birch.eecs.lehigh.edu' 'www-robotics.eecs.lehigh.edu'}
{'128.180.98.153'}
{birch 'birch.eecs.lehigh.edu' 'www-robotics.eecs.lehigh.edu'}
{'genie.eecs.lehigh.edu'}
{'128.180.5.9' '128.180.14.9' '128.180.98.9' '128.180.98.73'
                                                  '128.180.98.137'}
```

The association between Internet host names and IP addresses is maintained by the *Domain Name System* (DNS) [116]. SETL currently supports only the familiar *IP version 4* (IPv4) and not the emerging *IP version 6* (IPv6) forms of host names and addresses [194]. The DNS service is typically provided by a combination of information local to the host on which a particular request is made and knowledge maintained by *nameservers*. This is why the output of the above program when **ip_names** was applied to 'birch' was more extensive than when it was applied to 'genie', though the opposite was true for the application of **ip_addresses** because of the multiple network interfaces on 'genie'.

Without modifying impatient.setl, we can now implement a "blacklist" of clients that are to be denied service by the line-length server of Section 3.3.1. This, our sixth and final version of the line-length server, also shows how the HUP signal is conventionally interpreted by servers as a command to re-read configuration data. Here, it causes the server to re-read the file blacklist:

```
-- Line-length server, version 6 (prejudiced)

server_fd := open ('54006', 'server-socket');                    -- listen
```

```
assert server_fd ≠ om;                              -- or we crash

hup_fd := open ('HUP', 'signal');                   -- catch SIGHUPs

nasty := get_blacklist();               -- read set of names from database

loop                                                -- cycle indefinitely

  -- Await connection requests and HUP signals
  [ready] := select ([{server_fd, hup_fd}]);

  if server_fd in ready and
     (fd := accept (server_fd)) ≠ om then
    -- If client is not blacklisted, spawn background command
    if {peer_name fd, peer_address fd} * nasty = {} then
      system ('setl impatient.setl -- ' + str fd + ' &');
    end if;
    close (fd);
  end if;

  if hup_fd in ready then                           -- HUP caught
    dummy := getline hup_fd;                        -- receive the signal
    nasty := get_blacklist();                       -- re-read the set of names
  end if;

end loop;

proc get_blacklist();          -- read names of naughty clients from file
  return {id : while (id := getline 'blacklist') ≠ om};
end proc;
```

3.4 UDP Sockets

SETL provides support for UDP (the User Datagram Protocol). Although this is an "unreliable" protocol in that it does not include software mechanisms for retrying on transmission failures or data corruption (unlike TCP), and has restrictions on message length (a little under 65536 bytes), it is needed for applications that use broadcasting or multicasting, and it underlies such important applications as NFS (the Network File System), DNS (the Domain Name System), SNMP (the Simple Network Management Protocol), and various others noted by Stevens [194]. It is also likely to continue to figure prominently in some modern performance-intensive roles such as multimedia.

Strictly speaking, UDP is a "connectionless" protocol—a program can use a single UDP socket to communicate with more than one host and port number—but it is convenient for most UDP client programs to maintain the fiction that there is a connection, by keeping a local record of each server host and port number.

This is modeled in SETL by distinguishing between client and server UDP sockets, both in the way the first argument to **open** is specified and in the operations that are subsequently available on the resulting file descriptor.

To be a UDP client, a program makes a call very similar to what it uses when it asks to be a TCP client:

fd := **open** (*host_and_port*, 'udp-client-socket');

The host name and port number in the string form of the first argument here are separated by a colon, just as for TCP. UDP port numbers are entirely independent of TCP port numbers, though the IANA tries to register the same port number for both UDP and TCP when a given service is offered through both protocols. An integer representing an already open UDP client file descriptor is permitted as an alternative to the *host_and_port* argument, as usual. A successful UDP **open** makes available the operations

```
send (fd, datagram);
```

where *datagram* is simply a string, restricted in length as noted above, and

```
datagram := recv fd;
```

which receives a string into *datagram*. **Send** and **recv** are named after the Posix send and recv functions.

The following example program is the UDP analogue of the TCP client with which we began Section 3.1.1:

```
fd := open ('galt.cs.nyu.edu:13', 'UDP-client-socket');
send (fd, '');                -- send an empty datagram
print (recv fd);              -- receive and print a datagram
```

In the TCP case, opening the connection sufficed to prompt the server to return the desired information (the day of the week and so on), but when a UDP "connection" is opened, nothing is actually sent to the server. Even a null string will be wrapped in a UDP packet and sent by **send**, however, and that prods the server into action in this instance.

In fact, one of the most important practical differences between TCP and UDP is that there are no message boundaries in a TCP stream, whereas in UDP, every packet (datagram) is effectively a self-contained message, complete with a length that is implicit in its SETL string representation. For applications where reliability is not a concern and where all messages are known to fit within the limited size of datagrams, this can occasionally make UDP more convenient to use than TCP, though it is rarely the case that reliability can be so far ignored that it is acceptable for a program to sleep indefinitely waiting for a reply that never arrives, or to go into a confused state due to a message that has arrived twice. Both of these situations are perfectly possible with UDP, and the apparent convenience of UDP is but an evanescent illusion if the programmer has to write code to deal with them.

A UDP server socket is created in much the same way as a TCP server socket:

> *fd* := **open** (*port_number,* 'udp-server-socket');

Again, the port number specified to **open** is a string of decimal digits. The operations available on this kind of socket are:

> **sendto** (*fd*, *host_and_port*, *datagram*);

and

> [*host_and_port*, *datagram*] := **recvfrom** *fd*;

These are also named after the corresponding Posix functions.

Notice that the host name and port number must be specified afresh on every **sendto** call, and may be returned differently on every **recvfrom** call—the UDP server socket has no memory of any particular client after passing each datagram.

Following is the UDP analogue of the sequential TCP server of Section 3.1.2. Remarkably, it is even simpler:

```
fd := open('50013', 'UDP-server-socket');
loop
  [whom] := recvfrom fd;        -- ignore input datagram
  sendto (fd, whom, date);      -- send "date" datagram
end loop;
```

Because a UDP server socket can send and receive data, unlike a TCP server socket (which can only produce new connection sockets using **accept**), it can actually be used in a client-like role. The following program is functionally equivalent to the very first UDP client example above:

```
fd := open ('0', 'udp-server-socket');
sendto (fd, 'galt.cs.nyu.edu:13', '');
[−, datagram] := recvfrom fd;
print (datagram);
```

This rather subverts the notion of a client, but is interesting as an illustration of how the only fundamental difference between a UDP client and server in SETL is in the lack of memory that a UDP server has. This client in disguise obtains an arbitrary ephemeral port on the local machine and then uses the associated file descriptor to send an empty datagram to a UDP server which as usual replies with an information-bearing datagram. It is worth emphasizing that SETL maintains the distinction between UDP clients and servers only in that it restricts file descriptors opened as client sockets (according to the mode argument to **open**) to the use of **send** and **recv**, and restricts those opened as server sockets to the use of **sendto** and **recvfrom**.

3.5 Summary

This chapter has introduced the facilities that have been added to SETL for network communication. The liberal use of processes can be seen to be playing a role in the design methodology for handling various functional concerns that arise in the small but canonical example of the line-length server. **Select** has started to emerge as the central event-awaiting routine of servers and other processes that must be responsive at all times to nondeterministically arriving inputs.

In the next chapter, these themes are explored further in a larger server case study.

Chapter 4

WEBeye: A Case Study

The best way to organize any system, especially a distributed system, is to try to frame it as a hierarchy of command and control. In computer systems comprising a multitude of processes, most services will be provided by subroutines to their callers, child processes to their parents, and servers to their clients on the network. Even at the hardware level, the notion of a master and slave is common, and it is this tendency for active entities to play the role of coordinator or subordinate which transitively aggregates into the tree-like global nature of most systems.

At the top level, which is notionally unique even if physically replicated for survival reasons, is the genealogical head of a process tree. Client-server relationships induce further dependencies between processes, cross-linking the skeletal tree.

This kind of system has an inside and an outside (its interface), and I simply refer to it as a *Box*. The phenotype of a Box embodied in WEBeye will illustrate how the facilities described in Chapters 2 and 3 can work in harmony with the liberal use of fullweight processes to produce a system that is flexible, robust, efficient, and maintainable.

4.1 What WEBeye Is

The purpose of WEBeye is to allow browser users to aim and zoom a videocamera and simultaneously view the continuous stream of images being captured. The user can control the camera by clicking or dragging a pointer on either the image itself or small widgets adjacent to it.

The zoom widget is a slider with a red bar that follows the user's pointing actions and a blue bar that shows the amount of zoom actually achieved by the camera. The blue bar therefore tends to follow the red one at a rate constrained by the speed of the zoom servo-motor. For the Canon VC-C3 [35], the zoom factor ranges from 1 to 10.

The pan/tilt widget is the two-dimensional analogue of the one-dimensional zoom slider, and for the VC-C3 hardware has a range of -90 to $+90$ degrees in azimuth (pan) and -30 to $+25$ degrees in elevation (tilt), depicted on a flat grid. A red rectangle shows the requested position in pan/tilt space, and can be dragged or moved quickly by clicking, while a blue rectangle shows the progress of the hardware in moving towards the current goal. Additionally, the size of the current field of view is reflected in the size of the red and blue rectangles. This size is an inverse function of the zoom factor. More precisely, the width and height of the rectangles show directly on the grid how many degrees of azimuth and elevation respectively are subtended by the camera's current view, so that the user has an immediate indication of what part of pan/tilt space is being viewed.

The widgets for the real-time streaming image, the zoomer, and the pan/tilt control are simple Java applets which receive events from the pointing device (a mouse or equivalent) and from the WEBeye Box. Each applet maintains a rudimentary real-time graphical display and also sends events derived from user actions back to the Box.

Image production and camera control are managed by the Box, which is the "server side" of WEBeye. It notifies all clients of changes in the hardware parameters, and accepts requests to change those parameters. The Box also supplies clients with the processed image stream from the camera, which is independent of the camera control and motion event streams.

The Box services are provided through TCP streams. Some are designed for the direct convenience of Java clients, while others are more general in intent. Every service, except for those which supply compressed image data embedded in MIME documents, can be tested and learned about using the standard text-based telnet client. For example, the most comprehensive camera control service, which supports many commands, includes a help directive.

Besides Java clients, the Box supports browsers such as Netscape in a more primitive way by delivering JPEG images in either a streaming ("server-push") or snapshot mode. It also accepts camera control commands implied by "imagemap" clicks.

The httpd service provided by the vc-httpd.setl program listed in Section A.19 responds to HTTP requests as they might be generated by a browser, and instantiates a template document with port numbers and other parameters, such that the resulting document presents an imagemap with the live video playing in it. The hyperlink associated with the imagemap is a reference back to the httpd service, so that a "click" within the map causes camera motion and a new document instantiation. This is the only service in the Box that has both a video and a camera (motor) control aspect.

4.1.1 Video Services

From the client's point of view, the simplest video service is the "snapshot" service, snap. It sends a MIME-encapsulated JPEG image in response to any HTTP-protocol GET or POST request. A browser will always translate any URL beginning with the server's host name and port number into such a request, and display the resulting JPEG image as it is received. A non-browser client such as wget can use snap to take pictures periodically for archival, and Java clients use it to fetch images based on URLs in the same fashion as browsers.

Closely related to the snapshot service is the image-stream service, push. This is intended for browsers which support Netscape's "server-push" method of playing a continuous sequence of JPEG or GIF images contained in an indefinitely long multipart MIME document, and the httpd service just mentioned generates a reference to this service in the document it instantiates. For the sake of clients whose slow network connections could cause the buildup of images all along the route from server to client, the URL carrying this reference supports an optional *rate* parameter, measured (perversely) in milliseconds, which sets a minimum time between image transmissions. This rate can be included in a URL as *rate*=integer after a question mark, in the manner of parameters supplied by browser-based “forms” to Web servers.

Unfortunately, the only industry-wide standard for image streaming in browsers appears to be the one proposed for version 1.2 of the Java API, and most popular browsers at the time of this writing have only recently caught up with version 1.1 in terms of their built-in support and bundled Java classes.

However, a third “video” service in the WEBeye Box, giver, is offered as a transitional workaround to deal with this problem, which afflicts Java applets in both major browsers. A natural interface for Java clients is to have a simple command/response handshake over a sustained network connection. Each time the client requests another image by sending the command JPEG, the server replies with the latest image as soon as it has one that is different from the last one it sent that client. The server side of this has already been implemented, because just such a protocol is used by a server within the Box. The client side, if Java is used, is easy to implement in the 1.2 API version but practically infeasible with earlier versions. So, to make it as convenient as possible to adopt the new API when it becomes available and still provide a working (if sub-optimal) service meanwhile, the giver service accepts JPEG commands, but replies to each one not with an image but with a URL that the client can then use (even in the 1.0 API) to fetch the JPEG image itself. Each URL that this rather trivial service generates is a reference to the snapshot service, decorated with a sequence number to help defeat browser caching (which cannot even be fully turned off in Internet Explorer).

Finally, there is the image service within the Box which is used by snap and by httpd. It employs a command/reply handshaking protocol over a TCP connection, where the client receives a JPEG image in response to each JPEG command and the response is delayed until the latest image picked up by the server is different from the last image sent to that client, if any.

4.1.2 Camera Control Services

The pan/tilt/zoom camera control hardware accepts commands and delivers acknowledgements and event notifications over a serial line. The Box provides a bridge between this device and any number of Internet clients.

The most comprehensive of the camera control services, camera, supports a protocol that is designed to be convenient for programs and at the same time mnemonic for people, in the best tradition of Internet servers. Programmers intending to write code which communicates with the camera service can obtain all the information they need by using the standard telnet client to connect to the appropriate host and port number, which should begin a session with the server sending something like this:

```
>Welcome to the Canon VC- C3 pan/tilt/zoom camera control server.
>Type Help for help.  Cavil and whine to dB (bacon@cs.nyu.edu).
.
```

This will be displayed in the interactive telnet session window. If help is entered as suggested, the response will be something like this:

```
>
>Commands are:
>
>Help [command- name]
>Mode {Host | RC}
>Notify {On | Off}
>Zoom {[To] factor | By factor | In | Out} [At speed]
>Move {[To] pan tilt | By pan tilt} [[In] ms | At speed]
>{Up | Down | Left | Right} deg [[In] ms | At speed]
>Ramp ms
>Show {Mode | Notify | Zoom | Move | Position | Ramp}
>Clear
>Reload
>Setup
>Reset
>Quit
>
>A null command (empty line) repeats the previous command.
.
```

If the user then playfully types help help, the serious reply is as follows:

```
>
>Help
> -  Gives a compact synopsis of all commands, with optional
>    words shown in brackets [ ], grouping indicated by braces {
>    }, and alternatives separated by bars |.
> -  All command names and arguments are case- insensitive,
>    though for clarity they are shown here as literal names
>    starting with an uppercase letter.  Substitute a value for
>    any (possibly hyphenated) name that begins with a lowercase
>    letter.  Numbers may include signs and decimal points.
> -  Help is the only command besides Show which produces output
>    back to you, the client, when asynchronous notification is
>    off (see the Notify command).  You can tell where a piece
>    of help ends by where the ">" lines leave off and the final
>    "." on a line by itself occurs.  Server usage errors (your
>    protocol mistakes) are also reported in this "help" format.
>    Output from Show always consists of a single line, as does
>    each asynchronous notification (event message), so their
>    ends are also easy to recognize.
>
>Help command- name
> -  Tells you all about a specific command.
.
```

This describes the difference between help/diagnostic output and show/notify output.

Here are the results of help notify and help show:

```
>
>Notify On
> -  Turns on asynchronous notification.  You (the client) will
>    get an event message, formatted as a command recognized by
>    this server for convenience in playback, whenever there is
>    a change in the mode, zoom, pan/tilt, or ramp, and whenever
>    a zoom or pan/tilt limit is reached.  (Other messages,
>    with no corresponding command but formatted similarly,
>    will later be added.  For now, there is a catch- all message
>    "Canon", showing things the hardware is saying.)
>
>Notify Off
> -  Turns off asynchronous notification.  You can still get
>    information synchronously by using the Show command.
.
>
>All Show commands produce their output in the form of a
```

```
>command that could later be fed back in to the server to
>re- establish the state reported by the Show.
>
>Show Mode
> -  Yields Mode Host or Mode RC.
>Show Notify
> -  Yields Notify On or Notify Off.
> -  Each asynchronous notification (event message) and
>    Show result is sent to you, the client, on a single,
>    newline- terminated line.
>Show Zoom
> -  Yields the current zoom factor as a Zoom [To] command.
>Show {Position | Move}
> -  Yields the current pan and tilt angles as a Move [To]
>    command.
>Show Ramp
> -  Yields a Ramp command for the current ramp period.
.
```

Finally, here is the output from `help move`:

```
>
>Move [To] pan tilt [[In] ms] | At speed]
> -  Points the camera at pan degrees azimuth, tilt degrees
>    elevation, and stores these as the current values.
> -  Positive means right for pan, up for tilt.
> -  Range is - 90 to 90 for pan, - 30 to 25 for tilt.
> -  Resolution is 0.115 deg.
> -  The angular trajectory is shaped at each end by the
>    parabola suggested by the Ramp period.  If the angular
>    distance to move is large enough, maximum speed will be
>    sustained in the interval between the acceleration and
>    deceleration ramps unless constrained by the optional In
>    or At specification.
> –  If ”[In] ms” is specified, the server will try to plan a
>    camera motion trajectory that takes ms milliseconds.
> –  If instead ”At speed” is specified, the trajectory
>    speed will be limited to the given maximum during
>    the constant– speed interval between acceleration and
>    deceleration ramps.
> –  The units of speed in ”At speed” are deg/sec, with a
>    resolution of 1 deg/sec and a range of 1 to 70 deg/sec.
>Move By pan tilt [[In] ms] | At speed]
> –  Adds pan degrees azimuth and tilt degrees elevation to
>    the current pan and tilt values, and calls Move [To].
.
```

Now, suppose the user enters the command notify on, requesting event notification. Any updates to hardware settings resulting from, e.g., move and zoom commands will cause command–like messages to be sent to all such clients interested in events. For example, if this telnet user or any other client typed the command zoom out, a message such as

```
Zoom 6.180
```

would appear in the telnet user’s display. Clients can thus easily record camera control activity for later playback. Current settings can always be sensed with the Show command, so for example show move or equivalently show position might produce

```
Move To – 76.671 – 19.774
```

The camera service itself is little more than a command-processing front end for the do and notice services provided by the vc-do.setl program listed in Section A.11, which is responsible for maintaining a high-level, state-bearing model of the camera as an acceptor of commands and producer of events.

In principle, all clients could use the general camera service to issue commands and receive notifications, but it works out better if things are made even simpler for Java clients, which tend to be multi-threaded and therefore favor specialized services relating to their particular responsibilities. Accordingly, the jumper, mover, and zoomer services accept client input consisting purely of one or two numbers on each line, and build the appropriate command to send to the do service for each such line. Also in this category is the mouse service, which is so tailored to the needs of Java clients that they essentially just have to pass pointing device gestures through “in the raw”, and mouse maps these into camera control commands for transmission to the do service. This is effectively a generalization of the imagemap handling performed by the httpd service previously mentioned.

Similarly, there are Java-friendly services to provide an interface to the notice service, so that events can be delivered to Java threads in the simplest possible way. These are called evjump and evzoom.

The vc-do.setl program, in its event-producing capacity (the notice service), relies upon a pair of lower-level services, notify and event, which are linked such that every event received by notify is broadcast to all clients of event.

4.1.3 Administrative Services

WEBeye is designed with round-the-clock, unattended service in mind, but its administrative interface also makes it easy for people who are not computer experts to start, stop, and check up on the system, and to manage the log files.

WEBeye will normally run continuously unless there are problems severe enough to cause a critical component failure. For example, certain forms of resource exhaustion under heavy load conditions can trigger failures. The system is large and complex enough that it is also appropriate to "expect the unexpected" from hardware *and* software (see Section 5.3.3). When the unexpected happens, WEBeye does its best to bring itself down cleanly and completely, in the hope of clearing the condition which caused the failure. Then, when it is restarted, it has the best possible chance of survival.

In order to run unattended, WEBeye has to be able to be restarted automatically. This means that there has to be some program external to WEBeye that can observe when it has crashed or is not functioning correctly, and attempt to restart it. To this end, the vc-cron.setl program listed in Section A.9, which is intended to be run every minute from the administrative user's crontab file, steps through a series of checks on the presence and correct functioning of the Box. If the Box appears not to be running, or any of its principal servers fail to give satisfactory responses, vc-cron.setl tries to shut it down cleanly and restart it. However, in the event of recurring failures unrelieved by any indications of complete success, the frequency of restart attempts is decreased by powers of two until only one attempt is being made every 64 minutes. (This is in part an effort to avoid the Sorcerer's Apprentice syndrome, where an attempt to expunge one problem only leads to more problems, and in part an effort to avoid sending the WEBeye administrator too much e-mail, since a failure report is sent to that party every time vc-cron.setl tries to restart the Box.)

Restarting WEBeye involves shutting it down cleanly and then starting it again, as vc-restart.setl (Section A.37) does by calling vc-quit.setl (Section A.35) and then vc-go.setl (Section A.18). Another program, vc-check.setl (Section A.5) is also provided, and all these programs can be invoked via trivial wrapper commands such as restart for a more manual style of administration.

4.2 Software Structure

The primary program in the WEBeye Box is vc-toplev.setl, as listed in Section A.42. It is responsible for starting, monitoring, and stopping all the programs which provide one or more TCP services. These programs and their descendants can be clients of services in the Box, so vc-toplev.setl is careful to start and stop them in an order which respects client-server dependencies.

Following is a snapshot of a process tree taken a few minutes after the initialization of a WEBeye Box that is currently in operation [18] at Lehigh University. Each line begins with the number of minutes and seconds of CPU time consumed so far by its corresponding process. The indentation structure indicates parent-child relationships:

```
0:00 setl vc-toplev.setl
```

```
 0:03  \_ setlrun -5
 0:00      \_ setl vc-event.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-giver.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-image.setl
 0:44      |   \_ setlrun -8
12:56      |       \_ image-pump
 0:00      \_ setl vc-push.setl
 0:00      |   \_ setlrun -8
 0:11      |       \_ setlrun -8
 0:00      |       \_ setlrun -8
 0:00      \_ setl vc-snap.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-do.setl
 0:01      |   \_ setlrun -8
 0:00      |       \_ setl vc-model.setl
 0:01      |           \_ setlrun -9
 0:00      |               \_ setl vc-seq.setl
 0:00      |                   \_ setlrun -10
 0:00      |                       \_ setl vc-send.setl
 0:00      |                       |   \_ setlrun -11
 0:00      |                       \_ setl vc-recv.setl
 0:01      |                           \_ setlrun -11
 0:00      |                               \_ setl vc-input.setl
 0:00      |                                   \_ setlrun -12
 0:00      \_ setl vc-httpd.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-mouse.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-mover.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-jumper.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-zoomer.setl
 0:00      |   \_ setlrun -8
 0:00      \_ setl vc-camera.setl
 0:00      |   \_ setlrun -8
 0:00      |       \_ setl vc-ptz.setl -- 10
 0:01      |       |   \_ setlrun -11 -- 10
 0:00      |       \_ setl vc-ptz.setl -- 10
 0:01      |       |   \_ setlrun -11 -- 10
 0:00      |       \_ setl vc-ptz.setl -- 10
 0:01      |           \_ setltran
 0:00      |           \_ setlrun -11 -- 10
 0:00      \_ setl vc-evjump.setl
```

```
     0:00      |   \_ setlrun -8
     0:00      \_ setl vc-evzoom.setl
     0:00          \_ setlrun -8
```

At the time this snapshot was taken, there were two clients connected to the push image-streaming service (Section 4.1.1) and three to the camera control service (Section 4.1.2), though one of the latter had just connected—the instance of setltran indicates that we caught vc-camera.setl at a moment when it had just started to instantiate vc-ptz.setl as a child process for the new client, and the SETL translator was still active (setltran disappears immediately after passing the result of its translation to setlrun; occasionally, one can even catch an instance of setlcpp, the SETL preprocessor, in one of these displays).

Each negative number appearing in this process display is actually a command-line parameter passed from setl to setlrun identifying the file descriptor on which setlrun is to read the translated form of a SETL program. User-level command-line arguments to setl and hence setlrun are placed after an argument of the form "--", as mentioned in Section 2.1, and we see instances of this in the process display where vc-camera.setl passes the file descriptor for each newly accepted client to vc-ptz.setl. The fact that it is always the same number (10) here reflects the fact that vc-camera.setl always closes its own copy of that file descriptor after it has created the child process (vc-ptz.setl) which preserves the reference to the underlying kernel data structure, so the number is available to be used again in vc-camera.setl when the next client seeks a connection.

When vc-toplev.setl is started, it first attempts to make sure there are no other running instances that could conflict with it. The mutual exclusion mechanism is conservative. It is based on the existence of a lock file that is atomically created and destroyed. A clean shutdown of the Box ends with removal of the lock. If this does not happen, due to some catastrophic failure (such as loss of system power or an uncaught error in vc-toplev.setl itself), or if the lock exists because there is another running instance, the administrator of the Box is cautioned to perform certain checks before separately removing the lock and then restarting the Box. (See the *commence* procedure in vc-admin.setl, listed in Section A.1.)

Because of the dependencies between servers and clients through the services they respectively provide and use, including the dependencies transitively created by parent-child relationships, the next major job of vc-toplev.setl is to analyze all the source texts comprising the Box, so that it will be able to start servers in an order that ensures no client is started before a service it depends on is available.

This is a matter of preprocessing the SETL source programs, scanning for the recognizable patterns indicating the relationships of interest, and constructing the appropriate maps. Provided that consistent idioms are used to create child processes and to request TCP services, this is more reliable than trying to maintain a separate database of dependency information (a hazard familiar to Makefile users).

Dependencies on services effectively disappear once they have been satisfied by services becoming available, so just before starting to spawn the server processes, the top-level program sets up a little registry service to allow servers to “publish” their services, and a corresponding service for clients to look up information about services, particularly their dynamically assigned TCP port numbers.

With the registry in place, vc-toplev.setl proceeds to start servers which have no (outstanding) dependencies, waiting for all the services of the Box to be published, and shrinking the dependency maps as services appear. It also issues warnings if some services seem to be taking unreasonably long to publish themselves.

If and when all the services do come up, vc-toplev.setl announces to the world that the Box is ready for use by external clients. It does this by instantiating a template HTML document with an embedded URL that refers to a key port number in the Box. It then stores that document into a file and redirects a certain link to point to it. The link is what is called a *symbolic link* in Unix, which is a special kind of file that has no content of its own, but merely refers to another file, such that when input or output is performed on the link file, it really happens on the referred-to file. The link that is manipulated by vc-toplev.setl has a name that is known to a nearby Web server. This well-known link has four stages in its life cycle:

1. When the Box is cleanly down, it points to an HTML document which says so.

2. When the Box is in the initial process of analyzing dependencies and starting servers, the link is made to point to a static document indicating that the Box is in its “initializing” transition state.

3. When the Box is fully operational, the link is redirected to the HTML document that is instantiated with the main WEBeye port number as mentioned in the previous paragraph.

4. When the Box is about to attempt a clean shutdown, the link is again redirected to a static document identifying a transition state, this one called “closing”. Under normal circumstances, the Box will not spend long in the latter state, and the link will quickly be redirected to the “cleanly down” document.

There is actually one deliberate oversimplification in the above description, however. Because the principal interface to WEBeye is via the imagemap provided by the httpd service implemented by vc-httpd.setl, and this service already instantiates a template HTML document in response to each imagemap click, it is logical for the httpd implementation to serve the initial document as well. Redirection of a symbolic link is not an atomic operation in most versions of Unix, either—the link has to be destroyed and then re-created—so there is the slim possibility that it might not exist just at the moment a Web server attempts to read from it. For these reasons, the link is not read directly by the Web server itself, but by a CGI (*Common Gateway Interface*) script which waits a short while for the link to exist if necessary. This script then reads the file referred to by the link, and inspects it for a special prefix character sequence. If this sequence is not found, the file is presumed to contain a static document, which the script serves verbatim. If the special prefix is found, however, it is assumed to be followed by a properly delimited “host:port” designation of the lookup service. The script then uses lookup to find the host and port location of the httpd service, and opens a client connection to the latter. Web servers supply CGI scripts with any parameters that were received on the original request, typically arising from information in a URL after the script name, and transmit these parameters to CGI scripts through the PATH_INFO and QUERY_STRING environment variables. The CGI script that is generated at system configuration time from vc-master.cgi (see Section A.26) passes the URL-originated parameters through to the httpd service just as a browser would, and serves whatever httpd returns. Similarly, the CGI script that is generated from vc-jmaster.cgi (Section A.24) uses the lookup service to obtain the host and port locations of the services needed by Java-based clients, and substitutes these into the template HTML document it serves, which becomes the primary "page" for the continuous-motion browser interface featuring the control applets described in Section 4.1.

The template document instantiated by httpd refers back to the httpd service, so after serving the initial document, the Web server is not involved. Indeed, the Web server could have been bypassed entirely by a user who already knew the port number on which the WEBeye httpd service was listening, perhaps from previous contact or through having sufficient access to WEBeye's private files. In the case of the Java interface, the Web server is also involved only long enough to invoke the CGI script which instantiates the page referring to the applets, and to serve the byte code of those applets.

4.2.1 Video Services

The program vc-snap.setl, listed in Section A.41, implements the public snapshot service. It checks but ignores the details of the required HTTP GET or POST request from the client, and replies with a MIME-wrapped JPEG image, which it in turn obtains from the core video server.

Being a defensive public server, it deals with each client through a pump stream connected to a child process, and keeps a map from pump stream file descriptors to client information records. It is a small, multiplexing data processing module.

The program which implements the server-push service is vc-push.setl, listed in Section A.34. Like the snapshot server, it is not fussy about the details of the HTTP request. It does, however, support an optional *rate* parameter as described in Section 4.1.1. It is another small, multiplexing data processing module that deals with clients through child processes.

So is the vc-giver.setl program listed in Section A.17, whose giver service gives out URLs for a sequence of JPEG images as a temporary measure pending the widespread availability of the Java API that supports direct image streaming. Each URL it creates contains a reference to the snap (snapshot) service.

Finally, there is the image service provided by the vc-image.setl program listed in Section A.20. It is used by vc-snap.setl and vc-push.setl. It interfaces with an external program, image-pump, which is written in C and, when active, captures images and converts them to JPEG format as quickly as it can, telling the parent SETL server (vc-image.setl) about each image as it is ready.

This server, vc-image.setl, is intended for use strictly within the Box. It does a rudimentary check to authenticate each client (see the listing of vc-allowed.setl in Section A.2). Once satisfied, it dispenses with the usual need for an intervening child process and communicates directly with the client. It is not a complex program, but does keep a little state associated with each client so that each client can start receiving an image when it says it is ready to receive it and when a JPEG image that is new to that client has arrived from image-pump.

4.2.2 Camera Control Services

The camera service is provided by the SETL program vc-camera.setl listed in Section A.4. This program is really nothing more than a front end for a much larger program, vc-ptz.setl (Section A.33), which implements the protocol introduced in Section 4.1.2. The front end performs the important function of making the service available to an arbitrary number of clients simultaneously, which it does by instantiating vc-ptz.setl once for each client, as a child process connected through a pump stream. The front end server also logs some information, such as the beginning and end of each client session, and distributes TERM (“terminate”) signals when it receives a TERM signal, so that all the child processes will close their connections and exit rather than continuing after the camera service itself has (presumably) gone down.

When vc-camera.setl accepts a new client connection, it passes the new file descriptor to vc-ptz.setl on the command line. This child process then deals with the client directly through this file descriptor. In the interests of modularity and what Dijkstra called a “clear separation of concerns” [58], vc-ptz.setl does little more than parse and check protocol commands, and pass them to the local do service (see vc-do.setl) in the form of SETL maps. If the client of vc-ptz.setl has requested continuous asynchronous notification of events by issuing the command notify on, then vc-ptz.setl also maintains a connection to the local notice service, which (like the do service) is provided by vc-do.setl.

Once again, by the same structuring principles, vc-do.setl is a small program whose chief concern is the multiplexing of client sessions, and it hands the "real" work off to another program, vc-model.setl, which maintains a stateful, high-level model of the pan/tilt/zoom camera controller. Normally, vc-do.setl is instantiated just once, and so is vc-model.setl, for a given serial port. This model maintainer is strictly sequential, and completes each command (as passed down from vc-ptz.setl through vc-do.setl) before it begins the next one.

Thus vc-do.setl actually has to do some minor arbitration of requests from the multiple clients of its do service when they cannot all be satisfied at once. If it is busy (meaning that a command has been sent to vc-model.setl but not yet been replied to) when a command is received from a client, it will enter the request on a FIFO queue. However, each client is only allowed up to one pending request—vc-do.setl will not even read a further command from a client that already has one in the queue. Clients at this level get an explicit indication of command completion from the do service.

The jumper, mover, and zoomer services designed for Java clients, whose programs vc-jumper.setl, vc-mover.setl, and vc-zoomer.setl are listed in Sections A.25, A.29, and A.43 respectively, all use the do service in such a similar way that they are implemented by using the SETL preprocessor to define some symbols and then textually include vc-simpler.setl (Section A.40).

The gesture-interpreting mouse service, provided by the program vc-mouse.setl listed in Section A.28, also uses do.

The Java-friendly evjump and evzoom services, which are provided by the programs vc-evjump.setl and vc-evzoom.setl listed in Sections A.13 and A.14 respectively, use the notice service through the textually included vc-javent.setl listed in Section A.23.

The vc-do.setl program is the logical place to provide the notice service as well as the do service, because it is in direct contact with both the model maintainer and with a low-level event service, event, provided by vc-event.setl.

The vc-model.setl program, in its turn, implements high-level commands as sequences of lower-level commands, but it hands the actual timing, handshaking, and retrying responsibilities pertaining to command sequences off to yet another program, vc-seq.setl, which also happens to be in the best position to send low-level event notifications to a service called notify. The notify service is provided by the vc-event.setl program just mentioned, and works as follows: when a client apprises notify of an event, vc-event.setl distributes that event to all clients of its event service.

4.3 Summary

Complexity is the enemy of flexibility, robustness, and maintainability. Modularity is the best defense, and the free use of fullweight processes as modules has been illustrated in this chapter. These processes do not share memory, and only communicate by passing messages through pipes and over the network. This design restriction is not found to be burdensome in practice, and fits in well with SETL's value-semantics orientation.

The ease with which arbitrary numbers of clients can be tracked in a SETL program works hand in hand with this attention to modular design. This keeps all the programs in the WEBeye case study small and readable, supporting the Box's flexibility, robustness, and maintainability. Efficiency has been assured by isolating the CPU-intensive work in the image-pump program which is written in C and takes advantage of an existing JPEG conversion library. With this division of labor, we find it unnecessary to schedule programs for destruction by labelling them *prototypes*: the SETL programs already consume almost no resources compared with image-pump.

However, the interfaces between modules are not checked by an external party. There is currently nothing like CORBA's Interface Description Language (IDL) [153] for specifying what should and should not be in the maps that pass between SETL programs, and certainly nothing approaching the power of Ada specifications to pin down interfaces. The wise SETL programmer will therefore insert a few redundant checks on any received map. This is actually a very minor bit of housekeeping compared to the checking that needs to be done on data arriving from *external* sources on a network in a language-independent setting. In any case, good form in a piece of data is not usually enough to ensure that it makes sense when and where it occurs. The most subtle and extensive checks will be semantic and context-dependent.

Similarly, an exception-handling mechanism can be helpful, but does not relieve the programmer from considering exceptional cases. To the extent that processes which might reasonably be expected to raise exceptions are kept small and well isolated, the "safety net" effect for which most exception handling is specified is already achieved in the modular SETL designs presented here.

In the next chapter, we look at some more things the programmer can do to design robust, flexible, maintainable interfaces and programs for data processing over the Internet.

Chapter 5

On Data Processing

The world of externally motivated programs is much larger than that of elegant algorithms, and this semantic richness presents tremendous challenges to the language designer. It is is this kind of computing that I have been calling *data processing*.

5.1 The Field

In contexts as diverse as science, business, and academic research, data processing is as fundamental to what computers really do as computer science is to the study of what they might do.

But what is involved in data processing? All that usually happens to data when it gets “processed” is that it gets copied from one place to another. There is almost no computation in data processing, and few algorithms. And yet arranging for the right data to appear in the right place at the right time can be every bit as challenging as the most difficult mathematical analysis. More so, in fact, because the problems tend to be open-ended, and the psychology of users cannot usually be reduced to cogent proofs.

Neither does data processing have clear boundaries that distinguish it from other forms of computing, and we must be content with a rather intensional description of what it embraces. However, this thesis does not seek primarily to define data processing, but to show that SETL, conservatively augmented, is remarkably well suited to roles far from those described in *On Programming* [177].

Perhaps the most salient feature of data processing is that it tends to be concerned with data *interfaces*, spanning the range from low-level input/output formats to high-level user interfaces. A contemporary office information system will typically comprise mostly off-the-shelf software such as spreadsheet packages, database packages, word processors, and operating systems, configured for local use. Additionally, there will be customized elements such as graphical user interfaces and specialized programs relating to the specific activity of the organization. And then of course there is the data itself, which all the other components must directly or indirectly accommodate.

A major task of the data processing programmer is to provide interfaces between these components—decoding, observing data access coordination requirements, and, if not actually transforming data, then at least formatting it for presentation to one or more sinks. With the great rise in the importance of computer networks, the modern data processing programmer also has to be able to deal with the issues surrounding distributed concurrency, including latency and the need for redundancy and security.

We have come a long way from the simple, sheltered world of card readers, 9-track tapes, local disks, and line printers all operated in batch mode. The increase in the complexity of data processing environments is one reason why high-level languages are more important than ever. Fortunately, they are also more affordable.

5.2 Problems and Solutions

The intelligent modern programmer, faced with the task of designing a data processing system, will try to use existing packages as much as possible, and thereby reduce the job to one of coordinating large chunks of software by means of relatively small interconnection programs.

These programs, when they are written in an interpretive "shell" language, are often called *scripts*, and typically operate at the highest semantic level in the system, dealing as they do at the granularity of files and whole programs. Shell languages have their foundations in early "job control" languages.

Because the shell language has been the unavoidable starting point for any serious computer work right up until the age of the GUI (and still is, for many programmers), and because memory was not always a cheap and abundant resource, shell languages have tended to be very thin, lacking such amenities as lexical structure and sometimes even control structure. Many do not support the concept of an algebraic expression, and most provide little opportunity for static checking. Historically, therefore, the goal of a system designer has been to do as little as possible at this high level, and merely use the shell as a launching pad for programs coded for efficiency.

Not surprisingly, we feel the drag of this history today. People still tend (1) to use unnecessarily weak shell languages, (2) to use systems programming languages where high-level languages would be more appropriate, and (3) to show little regard for their shell scripts. The last of these phenomena probably results from the fact that most shell languages discourage good programming style. Every successful tool starts to be overused at some point, and this is exactly the situation we find ourselves in now—shell languages are being pressed to perform feats they were never designed for.

Much of this pressure on our worn-out tools comes from the modern data processing scene. What more obvious language to write small interconnecting programs in than the locally available shell language? If programs were as small as they are initially conceived to be in the minds of their creators, and if they stayed that way, all might be well, but of course the simple program hacked together in the weak language all too often grows into the unmanageable monster, and earns the respect of the unwary only after it has caused considerable frustration.

It is commonly said that the choice of language has a controlling influence on how we think about programming. It is equally true to say that it has a profound influence on what we think of a program *after* it has been written, though the operative factor in that regard is the care taken by the programmer to make the program readable in the first place. Of course, some languages make writing readable programs easier than others do.

SETL's particular contribution to readability, when programs are written in a style appropriate to its mathematical character, is that its most fundamental and reusable forms make sense when literally "read out loud" in phrases like "the set of [all] x in [the universe] S such that $P(x)$ [holds]" for "$\{x$ **in** $S \mid P(x)\}$", or "if [there] exists [an] x in S such that $P(x)$ [holds] ..." for "**if exists** x **in** $S \mid P(x)$ ...". These may seem idle matters to those who have no experience with SETL, but the "dual view" of sets and predicates alluded to in Section 1.1 is actually tremendously important in helping the programmer to stand back and look at a set or predicate as being delineated by constraints on a universe (the mathematical view), or to move closer and look inside to see a mechanism in which iterators produce candidate values that are tested in turn and either accepted or rejected (the algorithmic view). The strength in the fact that the same set or predicate can be viewed in *both* of these ways is that the mental image created for the one view provides a helpful doublecheck on the other.

A similar psychology obtains in the case $\{$*expression* : *iterators* | *predicate*$\}$, where further readability springs from the focus on the *expression* that characterizes all the members of the set. The singleton $\{$*expression*$\}$ then appears as a degenerate form of this set former, and the enumerated set $\{expr_1, expr_2, \ldots, expr_k\}$ is a natural generalization of the singleton, lending still more readability to SETL programs through uniformity of notation.

It is interesting to compare how SETL encourages high-level ways of thinking about problem solving to how languages such as Ada 95, with their strong support for defining and implementing high-level abstractions efficiently, do it. SETL takes the "minimum of fuss" approach, and really offers little beyond a few well chosen abstractions from the foundations of mathematics, generally free of inconvenient restrictions and machine-level concerns. Ada 95, on the other hand, deliberately predefines few abstractions, but provides facilities whereby a skilled programmer can create high-level abstractions running the gamut from generic to completely application-specific.

In the world of data processing, at least in that large part of it that involves small programs, SETL is attractive in combining readability with conciseness, so that a person with almost no knowledge of a system or its conventions can usually start to understand a SETL program rather quickly without getting lost in details. Of course, a well written Ada 95 program will also have this quality, but the program will probably have taken much longer to write than the equivalent SETL program, and will inevitably be longer textually. There is no place for the quick *and dirty* in any realm, but sometimes, especially for small programs, the writer's need to save time and the reader's need to get the right idea quickly are better served by a minimal, high-level script than by an exemplar of masonry.

If programs are the modules of a system, software engineering teaches us that we are most likely to achieve clean interfaces and comprehensible implementations by keeping those modules as small as they comfortably can be, too. In the modern data processing setting, the fact that a small program can be modified with much more confidence than a large one is also a good defense against shifting user requirements. Furthermore, the substantial cost advantages of using pre-existing large software components as much as possible dictates a strong anti-monolith policy in favor of small interconnecting programs. Finally, the rise of the network over the slowly rusting mainframe militates a distributed approach, and while Ada 95 is an excellent example of a language that allows the coordination issues to be dealt with even without sacrificing the advantages of static checking, Ada 95 is really a systems and applications programming language, as distinct from what might be regarded as a more than usually respectable scripting language (SETL). In a large data processing system, various languages, some of them quite specialized, will be found useful at various points, and again this argues for many little programs over a few big ones.

The tools at the disposal of a data processing programmer must be flexible, convenient, reasonably efficient, and robust. This is at least as much a matter of good implementation as it is of good language design. Because people tend to look down on data processing tools in the first place, they will rapidly become impatient with them unless they are obviously of high quality and scope, so although the design of a language should not be too fixated on implementation concerns, it should at least balance idealism with enough foresight to accommodate practice in a data processing context. This has been the motivation behind most of the SETL extensions described in this dissertation.

5.3 Guidelines

Let us now shift our attention from the responsibilities of the language designer and implementer to those of the programmer.

5.3.1 On Checking

Practically any data processing system will be forced to deal with some environment of “foreign” input data. It happens again and again that programmers, armed with tools better suited to systems programming than to high-level applications, will, in the face of deadlines, inexperience, and negligent supervisors, take shortcuts in the coding of input routines, and allocate fixed-size input buffers even in situations where they know they shouldn’t. This kind of bug remains dormant until some attacker or innocent button-pusher awakens it with a long input record, and when it finally bites by nibbling at some memory it isn’t supposed to, can be very difficult to track down. From this we learn the rule:

- No unchecked restrictions.

Actually, this should be refined a little, because overflows occur in many forms, and some of them are quite innocent. The correct advice is to be aware of when overflows are possible, and to make sure that their effects are understood (and not disastrous).

This rule is particularly relevant for distributed data processing, where programs tend to have greater exposure to malicious or clumsy adversaries than subroutines in relatively protected environments have. Stevens, in his famous introductory text on network programming, writes [194, p. 15]:

> It is remarkable how many network break-ins have occurred by a hacker sending data to cause a server’s call to sprintf to overflow its buffer. Other

functions that we should be careful with are gets, strcat, and strcpy, normally calling fgets, strncat, and strncpy instead.

The distinction he is making here is between ANSI C routines that respectively do not and do provide a way for the programmer to limit the amount of data written into a given memory area.

5.3.2 On Limits

Often the best way to guard against the ill effects of a memory or arithmetic overflow is to make sure the overflow can't happen at all. If the restriction isn't really necessary, perhaps it is worth removing, giving us the closely related rule:

- No silly restrictions.

This is entirely germane to the input buffer example, because the scrupulous programmer will either put in appropriate checks, or use some form of dynamic allocation to make sure there is always a big enough buffer if there is a buffer at all.

The truly wise programmer will have this taken care of automatically by using a language like SETL, which actually makes it more convenient *not* to have an input size restriction than to impose one. For example, if the input is organized into some kind of "lines" as defined by local file system and operating system conventions, the statement

line := **getline** *fd*;

assigns a single line (or **om**, at the end of file) to *line* no matter whether the line contains 0 characters or a billion. Its only alternative is to raise an exception due to insufficient virtual memory. The silent disasters of overflowing a buffer or yielding only part of the input line simply are not options, by the definition of **getline**.

Again, this is a rule which makes most sense at the highest semantic levels. At lower levels, closer to the hardware, some restrictions are unavoidable, and need to be properly checked. At the SETL level, it is usually most appropriate simply to pretend that no restriction exists, as in this example. This is really saying that exceeding the restriction is a rare resource exhaustion event that should probably be treated along the lines of running out of hard disk space—crashing the program with a diagnostic message may be a reasonable response, especially if the program is a small and well-isolated module whose main purpose is to deal with external clients, and its parent is a SETL program that sees the crash merely as an end-of-file condition on a file descriptor (see Section 2.13).

Language designers obey this rule when they pay allegiance to the principle of orthogonality. Some of my own extensions to SETL were made in this spirit. For example, any expression may validly initialize a **const**, general loop headers may appear within set and tuple formers, and **for** and **while** clauses can appear within a single loop header. Restrictions on such things as the length of identifiers have no place in a modern language design, of course.

Language implementers also do well to avoid things like fixed-size tables and incommodious integers wherever there is a risk of unnecessarily restricting program size, the number of symbols or procedures, and so on.

5.3.3 On the Unexpected

Every programmer makes mistakes, networks and computers crash, file systems overflow, resources of every description reach the point of exhaustion sometimes, and clients present a myriad of surprises, so:

- Expect the unexpected.

Except in safety-critical systems, which require a heavy investment in equipment and an entirely different approach to software design than what is appropriate for data processing (essentially to guarantee that every need is always covered by a working component, and that resource exhaustion absolutely cannot occur), the best way to maximize reliability in a large distributed system is to layer it, with each module doing local checks and fielding the failures of lower-level components.

When a check fails, a module's path of least resistance is usually to throw up its hands and fail completely but recognizably. The module at the next higher level, usually a parent process in a program hierarchy, should always be prepared to deal with such failure, if for no other reason that there are so many ways in which the child process (say) can fail. Design economy is achieved by routing various types of failure through a common handling mechanism. In the "Box" pattern embodied in WEBeye, this is done by having parent processes connected to child processes through pipe or pump streams. Failure or natural termination of a child process is made known to the parent through an end-of-file condition. The parent generally knows or doesn't care whether the termination was expected. Serious errors of the kind which "crash" the subprocess and evoke a complaint from a language processor's run-time system are also logged.

Note that the parent's responsibility here is usually to check for an end-of-file condition on each input operation, and take appropriate action, which may simply be to fail in turn. Even the end-of-file check can sometimes be elided if the parent is a SETL program that is content to crash immediately upon trying to use the **om** value ensuing from a failed read, though this is not very good programming style.

It is not that far, however, from the liberal use of the **assert** statement, which is to be recommended highly even though its only purpose is to abend the program in the ostensibly impossible case that the assertion fails.

Component failures can obviously lead to a cascade of failures, moving up the chain of responsibility. Child processes need to be aware that their parents can fail, and again the path of least resistance is usually for the child to exit when it sees an end-of-file condition on the communication channel with the parent. Sometimes a process will want to do some "cleanup" housekeeping before actually exiting. The *exit_gracefully* routine in the vc-exit.setl file (Section A.15) that is textually included by many components of WEBeye is a fairly extreme example of this, as it propagates SIGTERM signals to subprocesses in an effort to give them all a chance to shut down cleanly, but ends up issuing the irresistible SIGKILL to any that do not respond to the SIGTERM. Conversely, processes in WEBeye that have children strive to remain receptive to SIGTERM at all times, as is perhaps best illustrated by the use of a routine called *select_or_exit_on_sigterm* throughout vc-seq.setl (Section A.39).

At some level, in a good design, a cascade of failures will reach a module which attempts some form of recovery. The main advantage of *not* being overzealous in riddling lower-level modules with recovery code relates to the variety of possible failure causes in real systems: unless the failure has a very specific and immediately recoverable cause, the best chance for a "clean slate" upon which to bring the failed part of the system back up will be engendered by clearing out the failed incarnation as completely as possible. This is especially true if the root cause of the failure was resource exhaustion, one of the most unpredictable and problematic failure modes—the very act of removing a large subtree of processes in such circumstances may be the most important part of the recovery itself, as it frees up a large quantity of vital resources.

5.3.4 On Clients

The sequence of server examples presented in Sections 3.2 and 3.3 implicitly suggested the rule:

- Never trust clients.

Often it will be the case that a child process appointed by a server to deal with a client will have numerous responsibilities, and protection against denial-of-service attacks through resource exhaustion can simply be part of the child's natural propensity for crashing on conditions it cannot handle.

But sometimes the child process is interposed purely for protection at the communications level. Let us now examine a couple of general-purpose child processes that can be used by any server for safe line-by-line communication with any client.

The model, as always, is that it doesn't matter if the child process crashes, but it matters very much if the server crashes or is blocked in an I/O operation other than its main **select**. Hence the server should not even try to read something as seemingly innocuous as a single line directly from a client, because the client could send part of the line and then pause indefinitely. Similarly, the server should not send a line directly to a client, because the client could absorb part of it and then block.

To handle the input side of a connection to a newly accepted client having file descriptor *fd*, the server can use the following trivial program, which merely copies lines from **stdin** to **stdout**, flushing **stdout** after each one:

```
tie (stdin, stdout);                -- auto-flush stdout on each read from stdin
while (line := getline stdin) ≠ om loop      -- read line or EOF indication
  putline (stdout, line);                      -- write line (and auto-flush)
end loop;                                     -- loop ends when EOF reached
```

If this program is named line-pipe.setl, the server can start it as an intermediary between itself and the client as follows:

```
fd_in := open ('exec setl line-pipe.setl <&' + str fd, 'pipe-in');
```

Notice that the shell has been used to redirect input from *fd* into the child process's **stdin**. Now whenever this child receives a *whole line* from the client, it writes it out to its own **stdout**, which is connected by a pipe to the parent's *fd_in*. Whenever the parent (the server) is ready to receive input from the child, it can read the whole line at high speed through this pipe. Typically, *fd_in* will be one of many file descriptors in a set passed to **select**, and the child's attempt to write the line to its parent will cause **select** to wake up and return *fd_in* in a set of "ready for reading" file descriptors.

On the output side, matters are not so simple. One might reasonably expect that the same little program could be started by the parent using

```
fd_out := open ('exec setl line-pipe.setl >&' + str fd, 'pipe-out');
```

since if the parent waits for *fd_out* to become "ready for writing" before sending it a line, then it can do so in the certain knowledge that the child will accept the whole line at high speed, no matter how long it takes for the child to send that line along to a slow (or even indefinitely blocking) client.

But here we run into an annoying fact about Unix pipes (a fact which is usually quite welcome from the performance point of view): that they can be filled "to capacity" by a sender in advance of the receiver being ready to receive even one byte. The result is that from the point of view of senders, receivers appear to be ready to receive before they really are. In the present case, this means that line-pipe.setl may appear to be ready to receive a line of data from its parent when in fact it is in the middle of doling a previous line out to an arbitrarily slow client.

A solution to this problem is to use a sending child process which gives its parent an explicit indication when it is truly ready to receive a line:

```
fd := open (val command_line(1), 'w');      -- get fd from command line
```

```
tie (stdin, stdout);              -- auto-flush stdout on each read from stdin
while (line := getline stdin) ≠ om loop     -- read line or EOF indication
  putline (fd, line);                          -- write line to remote client
  flush (fd);                                   -- flush client output buffer
  putline (stdout, '');                     -- tell parent we're ready for more
end loop;                                      -- loop ends when EOF reached
```

If this program is named line-pump.setl, the parent can invoke it thus:

```
fd_out := open ('exec setl line-pump.setl -- ' + str fd, 'pump');
```

Notice that *fd_out* here is actually a bidirectional file descriptor, and that *fd* is not redirected by the shell but instead identified on the child's invocation command line and inherited. The parent must now wait for *fd_out* to become ready *for reading* and clear that indication by reading the empty line from *fd_out* before sending any line (on *fd_out*) after the first one. (An obvious slight variation on this parent-child protocol is to have the child send the parent an empty line initially as well, so that every write by the parent, including the first, has to be preceded by absorption of the empty "clear to send" line.)

Once the parent has opened both *fd_in* and *fd_out*, it is free to **close** (*fd*). The actual network connection will not be closed until both child processes have released it, which a cursory inspection of line-pipe.setl and line-pump.setl shows will not happen until they both terminate.

When a process tries to write through a pipe or pump stream to a process that has terminated, a PIPE signal is sent to the would-be writer. This signal can also be generated by attempted output to a TCP connection that has been closed by the peer, though the semantics are a little more complex (the error indication will only be generated after the *second* low-level write to such a stream). Normally, SIGPIPE causes silent termination of the process, though this behavior can be overridden in the usual way through a call such as

```
open ('SIGPIPE', 'ignore');
```

The process which does this should also normally check for errors on all output operations to pipes, pumps, and network connections:

```
clear_error;
putline (fd, ...);
flush (fd);
if last_error $\neq$ no_error then
  -- output error has occurred
    ⋮
```

Clearly, this is a messy business, and best avoided.

Another advantage of using line-pump.setl, in addition to endowing servers with a simple output flow control mechanism, is that it assists servers in just such avoidance—it eliminates their need to consider PIPE signals explicitly. If the child process goes down on a SIGPIPE, the parent merely receives an end-of-file indication from the child. The situation we have here is that the parent (server) always waits for the child to declare its readiness to receive a whole line. The child will not at that point be trying to write to an adversarial client, so it will not itself cause a SIGPIPE to be sent to the parent.

Let us now complete this picture. The parent that is communicating with a client through the two processes just reviewed should check for an end-of-file condition on *fd_in* whenever it becomes “ready for reading”. The same is true for *fd_out*, which is bidirectional despite its name. This can be done in the usual way such as by checking for an **om** return from **getline** or by interrogating **eof** after a **geta**. An end-of-file from *fd_in* will usually mean that the client (or its host, or the network) has closed the connection. An end-of-file from *fd_out* is less likely to be normal behavior, but essentially means that a dropped connection has led to the line-pump.setl child terminating on a SIGPIPE. In both cases, the parent can finish by executing the following code:

```
kill (pid (fd_in));          -- send SIGTERM to input subprocess
kill (pid (fd_out));         -- send SIGTERM to output subprocess
close (fd_in);
close (fd_out);
```

One or both of the **kill** calls will be redundant but harmless here. The case where a **kill** is not redundant is where the client has left one side of the connection (input or output) open but blocking. The **kill** makes sure that the child process is not trying to complete an input or output operation while the parent is left waiting for it to exit—**close** on a pipe or pump stream involves a low-level wait, which can be indefinite if the child is blocked.

5.3.5 On Aliases

Memory management, even when low-level allocation is hidden from view, is always an issue: one of the decisions a programmer repeatedly has to make is whether to copy or merely to reference. Most languages make it easier to reference than to copy. This is only natural considering that languages have traditionally been designed from the machine upwards, because it usually takes less CPU time to copy a pointer than to copy the data it points to. But the effect at the application level is that programmers tend to code for copying only if it seems necessary.

And as a result, they all too often produce "pointer spaghetti" which ultimately leads to bugs in which aliases are mistaken for unique pointers, blocks are deallocated prematurely, and pointers fail to get updated when their referents are moved.

SETL, on the other hand, encourages copying with its so-called "value semantics". There is no way to create an alias in SETL in the usual sense of more than one variable referring to a single object, nor are there pointers *per se* in SETL. Assignment, including both directions of parameter passing, is defined as a full copy of the object regardless of whether it is a simple number or a vast and complex map. (Of course, implementations are free to optimize out the actual copying, using, for example, a copy-upon-change regime.) To emphasize this orientation, I usually speak not of SETL objects, but of SETL values, except where it is actually necessary to distinguish between a value and its machine representation. The closest thing to a pointer in SETL is a value that serves as a "key" or domain element in one or more maps. Each map plays the role of a memory, and the map's name has to be mentioned on every fetch or store. Because a value of any type can be a key, maps are fully associative memories with unbounded address (key) spaces. Where the key space is naturally a dense set of small positive integers, a tuple can serve as the map/memory.

In short, SETL pushes programmers gently but firmly in the direction of the salutary rule:

- No unnecessary aliases.

And when aliases *are* necessary, SETL insists that they be referenced to specific maps.

The effect on programs of this bias is far-reaching, and likely to be somewhat discomfiting to people with a LISP background. It is a major paradigm shift. Whereas LISP focuses on the map element (the ordered pair, or "cons" cell), SETL treats the whole map, and does so with considerable regard for human syntactic needs. This represents a significant elevation in the semantic level, which is perhaps not surprising—pure LISP is, after all, nothing more than a machine language for a tiny recursive interpreter.

This "anti-alias" recommendation, however, though I believe it to be highly appropriate for virtually all data processing programs, is not always good for systems programming. It is hard to imagine a tree manipulation package or operating system kernel written in C without pointers or Ada without access types.

5.3.6 On Accessibility

Avoiding bottlenecks, and providing helpful redundancy in the form of doublechecks and assertions, as well as the very sound rules about modularization, abstraction, and even style that have emerged from the young science of software engineering, are all just as valid for applications programming and systems programming as they are for data processing, but there is another, much humbler rule which is particularly worth following in the specific context of data processing:

- No unprintable data.

In other words, all data that passes between data processing programs should be represented in a form that will be displayed by the most basic tools such as text editors and printers in "natural" denotations, unless there is some compelling reason for not doing so. There was, a long time ago, some justification for "binary" formats, which can save CPU time, disk space, and communication bandwidth, but as of well before the 1990s, these are trivial, inconsequential benefits at the data processing level when weighed against the inconvenience of data that can only be viewed through special filters. Of course, wherever formats are predefined, this rule cannot necessarily be followed, and insofar as browsers are now basic tools for displaying data, this rule does not necessarily mean that everything should be constrained to the "printable" part of ASCII (strictly speaking, the print class of characters in the POSIX locale defined in Unix 98 [154]), though this is probably still desirable for all but image data (and even for images is sometimes the best choice).

SETL actively supports the bias in favor of printable data. It has a good repertoire of facilities for deciphering and formatting arbitrary values as strings (Section 2.14.2), it features the **pretty** operator (Section 2.14.3) which produces printable strings exclusively, and the general output routines render values legibly except that they do not interfere more than necessary with the contents of strings. Routines such as **getchar**, **getfile**, **putchar**, and **putfile** do not interfere with them at all, ensuring that any bit pattern can be read or written if necessary. In the case where pure SETL programs are exchanging data, **writea** and **reada** can be used as perfect reciprocals, making the data stream an ideal place to probe for testing or instrumentation purposes. Similarly, if programs are designed to communicate using a line-by-line protocol, typically incorporating some simple command language, **printa** can be used for formatting and sending messages to a corresponding **geta**, which will receive each message as a whole line before parsing and interpreting it. This mode of operation is especially appropriate for communication with servers, which usually need to be able to defend themselves against miscreant clients. A primitive tool such as the general telnet TCP client can be used to perform some basic tests on such a command-oriented server.

5.3.7 On Program Size

Because even large, complex data structures are just values in SETL, they can be passed from one SETL program to another with consummate ease by the primitive **writea** and **reada** operations just mentioned. This further facilitates the division of labor into many small programs instead of a few large ones, and hints at the rule:

- No monster programs.

The nature of modern data processing, to the extent that it involves piecework done by programmers with a flexible attitude towards languages and configurable off-the-shelf software, pressures programs to be small. Conversely, the affordability of large populations of processes on modern hardware removes the efficiency obstacle to treating programs as a plentiful resource. Furthermore, the relatively high walls of protection provided by modern operating systems around processes suggests that programs themselves may be ideal as modules or even “objects”. Indeed, what practitioners of object-oriented programming now speak of as a method call was originally defined literally as a message-passing operation [185, p. 438].

There are numerous advantages to the use of programs as the fundamental modules in data processing systems. First, each program can be written in whatever language is most appropriate for it—even “call-out” conventions usually constrain the choices severely in the single-program case. Second, independent threads of control help to avoid bottlenecks, and to ward off the syndrome of a single program trying to juggle multiple activities. Third, shared resources tend to be guarded by their own supervisory processes rather than being carelessly managed by global variables (though of course a careful programmer would make such things private to a package and accessible only indirectly through subroutines). With regard to the latter, the motivation to copy rather than reference data is clearly strong in the setting of “fullweight” processes—resources will only be shared if they need to be.

5.3.8 On Standards

It goes without saying that adherence to recognized standards such as the Internet protocols and HTTP/MIME is a compatibility prerequisite for practically any new piece of software that hopes to deal with the global public network, and that confining it to use the API, shell, and utilities defined by Unix 98 [154] and Posix [118, 119, 117] where feasible will lend it a high degree of portability.

But there are also some other specific rules which should always be followed unless there is some compelling reason not to. These are rules that have become established practice because they work well.

5.3.8.1 Port Number Independence

As recommended in Section 3.1.3 and illustrated in Section 4.2, servers should strive to be independent of any specific TCP or UDP port number. To do otherwise is to risk making it impossible for a service to be offered at all, which will happen if the port number is already in use by another program. This condition can persist indefinitely, as is likely if the other program is itself a server. If a server critically depends on obtaining a certain port number, and some fundamental servers do (e.g., Web servers), then the port number should at least be registered with the IANA [116], though even this is no guarantee of its availability. Such a server's chances of getting the port number it wants will be further improved by having it started soon after its host comes up, perhaps as part of the system initialization sequence.

5.3.8.2 Configuration and Installation

A little effort on the part of the software developer to make a package easy to configure, install, and maintain can save every person responsible for installing and administering it a significant amount of trouble and vulnerability to mistakes. This is true for any software package, but especially so for large and complex systems that require configuration decisions to be made by the installer.

In this regard, perhaps the most important rule is to provide a step-by-step installation procedure that offers reasonable defaults and an opportunity to override them, together with clear documentation on the places where the software package impinges on the target platform. An installation script can be quite helpful.

A good principle to follow is to try to minimize, within reason, the number of dependencies on specific files or other resources in target systems. So, for example, although a software package may comprise a large number of programs and configuration files, they should by default all be grouped under a directory whose name serves as a common prefix, if practicable. Then this prefix, together with any particular system files that need to be inserted or modified, will be the entire extent of the package's "footprint" (apart from the space and time it ultimately consumes, of course).

One convention deserves special mention where servers are concerned, and that is the matter of how to make them sensitive to configuration changes without requiring them to be stopped and restarted. Fortunately, the Unix tradition has an answer to this question: make the server accept SIGHUP (the "hang-up" signal) as a request to re-read the configuration data. For example, inetd (the so-called "super-server" that is running on virtually every Internet-aware Unix host) re-reads its configuration file, /etc/services, whenever a HUP signal is sent to it.

For a server structured as an event loop, as in Section 5.3.8.3, this behavior can be easily implemented by including a HUP signal stream among the file descriptors passed on the main **select** invocation.

5.3.8.3 The Event Loop

Any SETL program that waits nondeterministically for inputs from more than one source will do so by calling **select**. For example, even the simple server vc-snap.setl listed in Section A.41 and discussed in Section 4.2.1 is typical in maintaining a map from pump file descriptors to client records. Each pump stream is connected to a child process that deals with one particular client. The domain of the map, i.e. the set of pump file descriptors, is passed to **select** along with the the file descriptor of the socket that listens for new client connection requests. Again, this is an entirely typical arrangement, where the server delegates all long-term work to subprocesses and gets back to its main job, sleeping in a **select** call, as quickly as possible. If the server had other events to be concerned about, such as HUP signals telling it to re-read configuration data, or timers telling it to do some periodic checks, the file descriptors for those signal or timer streams would also be included in the set passed to **select**.

Personally, I find myself most comfortable with **select** appearing naked in an overt main event loop, but the sensibilities of those who prefer the “callback” style of programming can easily be accommodated too. Suppose *fd_map* is a global map from pump file descriptors to records, each of which contains a *handler* field designating a unary event-handling routine, and *fd_ready* is a global set-valued variable. Then the SETL main program, if the programmer so wishes, can consist of nothing more than some initialization and a final call to a routine such as

```
proc process_events;
  var fd;     -- local
  loop     -- cycle until some event-handling routine executes a stop
    [fd_ready] := select (domain fd_map);
    -- The set fd_ready is rechecked on each iteration:
    for fd in fd_ready | fd in fd_ready loop
      call (fd_map(fd).handler, fd);     -- indirect call
    end loop;
  end loop;
end proc;
```

which could be incorporated verbatim into programs using **#include**, preceded if desired by **#define** lines that rename *fd_map* and/or *fd_ready*.

“Registering” an event-handling routine named (say) *client_input* could be done with

```
register (fd, routine client_input);
```

where the procedure *register* is defined as follows:

```
proc register (fd, callback_routine);
  fd_map(fd) ?:= {};     -- establish new record if necessary
  fd_map(fd).handler := callback_routine;
end proc;
```

"De-registering" a callback routine via

```
deregister (fd);
```

might then be done by the following procedure:

```
proc deregister (fd);
  fd_map(fd).handler := om;  -- caller may now remove whole record
  fd_ready less:= fd;         -- remove fd from transient ready set
end proc;
```

The usual cautions about manipulation of global variables apply here: callback routines must be sensitive to what other callbacks might do to those variables. This is why *process_events* has the odd-looking loop header "*fd* **in** *fd_ready* | *fd* **in** *fd_ready*", which inspects the same global *fd_ready* set as *deregister* modifies. This loop header makes sure that each *fd* produced by the first "*fd* **in** *fd_ready*", which iterates over a *copy* of *fd_ready*, is still to be found in the global variable *fd_ready* before the corresponding loop iteration occurs. If it is not there at that time, a previous iteration has de-registered the file descriptor and its associated event handler from *fd_map*, and it would then be inappropriate to try to call that event handler.

It may appear at first glance that this circumstance could be dealt with more gracefully simply by guarding against *fd_map*(*fd*) or *fd_map*(*fd*)*.handler* being **om**, but this would offer no protection against the case where a file descriptor, retired by both a call-back and by a **close** executed by that callback, reappeared on a subsequent callback's **open**—indeed, Unix will always yield the most recently closed file descriptor on a system-level open call. This file descriptor could then be mistaken for its older incarnation, and inappropriate processing performed on it. Since it is really a new file descriptor that has not yet entered the set of candidates supplied to **select**, the appropriate processing for it is none at all, at this point. Notice that performing "*fd_ready* **with**:= *fd*" in *register*, perhaps in a fatuous appeal to symmetry with *deregister,* would be exactly equivalent to making this oversimplification.

In WEBeye, where each main server loop typically has a **select** call over at least the domain of a *clients* map and a listening server socket, these semantic subtleties are kept at bay by adhering to the principle that in the code following the return from **select**, operations which may shrink the *clients* map precede those which may expand it. For example, tests for input from existing clients, which cause shrinkage of the *clients* map when clients terminate their connections as indicated by an end-of-file condition on the input file descriptor, are placed before the test for newly connecting clients, which can increase the size of the *clients* map.

5.4 Summary

This chapter has tried to make the case for the use of a high-level language in Internet data processing, by enunciating what I feel to be useful rules of software design for this kind of environment and by indicating how SETL, in particular, supports a software engineering methodology which obeys them.

Chapter 6

Conclusions

The strengths of SETL as a data processing language follow largely from its properties which (1) make it a good language for high-level algorithm description and prototyping, (2) encourage adherence to the programming guidelines given in Chapter 5, particularly the one which advocates employing plenty of processes, (3) reflect typical data processing environments through a convenient predefined interface to the most commonly used parts of Unix 98, and (4) allow SETL to serve as a highly competent "glue" language [156] for interconnecting programs written in other languages.

The case study of Chapter 4 has shown how the general-purpose SETL I/O facilities, operating system interface, and miscellaneous extensions described in Chapter 2 work with the sockets-based network support described in Chapter 3 and the modularity of small, high-level programs to produce a system that is comprehensible, robust, and maintainable.

In this concluding chapter, I will review a few other such systems, and then discuss some needs which have become apparent, particularly in the area of types.

6.1 Other Systems

I have written many server systems in SETL to date. Most of them have been small and even trivial, which is as often desirable in a server hierarchy (Box) as in any other system. Others have necessarily been larger.

For example, one Box which is of intermediate size is the PWM Toolkit. Its "business end" sends commands on a serial line to a program in a PC-cum-microcontroller which generates effective-voltage control signals on any subset of the pins of a parallel port by direct pulse-width modulation (PWM), i.e., by carefully timed, very rapid toggling. It has been used to drive the spherical pointing motor [26, 25] through an H-bridge switch.

The PWM Toolkit supports several possible methods and modes of signal generation, some of which involve dependencies between pins. Besides controlling the various basic quantitative parameters of PWM, the toolkit can generate time-varying envelopes, including some cyclical patterns with their own parameters such as maximum, minimum, frequency, and phase.

The PWM Toolkit may be used by telnet clients (it has a command-line interface with a help command) or more conveniently by a GUI client, which uses the command-line interface to the server but presents to its user a master window and zero or more detail windows that depend on selections made through the master window. The windows themselves are all created and governed by a wish shell process that reads dynamically generated Tcl/Tk [184] command scripts.

The widgets created by these scripts are originally specified by templates consisting of nested tuples in a SETL program, where the horizontal or vertical layout of GUI elements at a given nesting level is controlled by whether the level is even or odd. The wish process is attached to the SETL program through a pump stream. The Tcl/Tk commands are issued by the SETL program on this stream to build, destroy, and update widgets based on inputs from the user and from the PWM Toolkit server. Each widget, whenever the user manipulates a control, sends information to the SETL program on the pump stream by writing a short message to what the widget sees as the standard output stream of the wish process—this is the main way the SETL program receives user input, although it also keeps a debugging stream open for users to enter arbitrary wish commands, which it merely passes along. Web clients using an interface known as *LogEye* [17] to view a foveated “log-map” image also implicitly communicate with the PWM Toolkit server when controlling the effective pan and tilt voltages supplied to the spherical pointing motor on which LogEye’s miniature videocamera is mounted. All clients which have issued the notify command to the server receive notice of all parameter changes. The GUI clients automatically issue this command on startup, which has the interesting and useful effect of causing all the sliders and other controls associated with these clients to change state automatically (in real time, from the user’s point of view) when other clients modify parameters through their controls. It is even possible for two users to get into a tug of war by dragging similar controls in opposite directions. This keeps the server rather busy changing parameter values, but causes no real harm except perhaps to a hardware device that has difficulty with rapidly changing effective voltage levels.

The LogEye system just mentioned and another Web-interfaced service I wrote called *LabEye* [16] are in some respects similar to the WEBeye of Chapter 4, and antedate it. They cannot really be described as simpler than WEBeye. LogEye, for example, has the ability to manage, filter and cache a variety of live and stored image streams simultaneously, and features a calibration procedure for interpolating the PWM values that should be used for aiming a videocamera based purely on convolving image samples corresponding to various PWM settings. LabEye allows the user to control and view patterns on an oscilloscope and two bi-color LEDs using a browser interface to a server which indirectly commands a BASIC Stamp [167] that in turn controls a circuit of my own construction. These systems have acted as proving grounds for many of the SETL-based, process-intensive techniques outlined in this dissertation.

Richard Wallace has also used the Box approach in the SETL prototype of his ALICE [206] artificial-intelligence conversationalist. Its primary interface was through a Web browser which communicated with a server written in SETL. That server spawned a child process to read client input and send MIME-wrapped HTML code in reply, just as the httpd interface of WEBeye does. In the best modular tradition of Boxes, the original ALICE made use of filters and pump streams invoked from SETL, and of another server which did the main work of natural-language processing. Some user input strings were interpreted as commands to move a videocamera, so ALICE also played client to the camera-control server side of LogEye, and embedded a reference to the video server aspect of LogEye in the HTML sent to the client browser, in order to serve the user a picture (usually a live one), a textual response, and a prompt for more input.

The idea of embedding, in the HTML sent to a client presumed to be a browser, a reference back to the server which generated that HTML (or at least to a server which is in turn a client of that HTML generator) is used by WEBeye, LogEye, LabEye, and ALICE. In fact, this kind of self-reference, in an imagemap, was used in the original LabCam at NYU. It is similar in spirit to the use of a CGI script, but bypasses the need for a full-fledged Web server to be involved at every step of what is often a fairly long-term interaction. The general principle is appropriate whenever a Web page is designed to lead the user to a similar page one or more times.

My use of SETL for processing Web-based requests began with CGI scripts, and I feel it is still useful in that role. For example, the rudimentary comp.lang.ada interface service [14] I wrote when visiting Alfredo Ferro and his colleagues in Catania in 1994 was not only moderately useful in the rather speed-limited network environment that prevailed there at that time, but was also one of the earliest tests of “sockets for SETL” as I termed the first, TCP-only version of the current set of SETL library extensions for network programming.

The “Famous Original” SETL Server [13] that has been accessible through my home page [15] for several years is not actually a server at all in the TCP/IP sense, but simply a rather general CGI script which allows any user to run a SETL program by entering it directly into a text sub-window on a Web page or by giving a URL at which the program can be found. Similarly, run-time inputs to the SETL program can be supplied in either of these two ways. Since the SETL program runs on the Web server host or a delegate thereof, it is run in a restricted mode which prevents server-side abuses such as the clobbering of arbitrary files. The administrator of the SETL Server can allow guest programs some latitude in opening client sockets, however, by configuring a set of host:port combinations that foreign programs are allowed to connect to. This is mediated by the support for the secure restricted mode that is built into my SETL implementation.

6.2 Interoperability

By far the most important kind of interoperation between code written in different languages is that which occurs when the the pieces of code are in fact entirely separate programs, linked only through some communication medium. This is especially true when processes are an abundant resource within easy reach, and is such a powerful and general *modus operandi* that I find it tends to remove most of the need for subroutine-call interoperability between high-level languages like SETL and lower-level languages like C.

This is largely because of the differences in data representation between languages of unequal level. These differences require data conversions, which introduce the potential for error, inefficiency, and clutter. The conversions must occur even if the interface is call-based rather than I/O-based. Conversely, if interfaces are kept narrow where practicable as a fundamental design principle, the high walls between processes are a welcome form of protection: memory corruption by a SETL program running under a correct interpreter is impossible, but no such guarantee can be made if an arbitrary C library is linked in. Furthermore, where a language split is already countenanced for the sake of access to some optimized low-level code, a process split allows the CPU-intensive computation to be moved easily to a different processor, perhaps one that is much faster than the user's workstation.

However, there are occasions when it is genuinely useful to be able to use a predefined library of C, Fortran, or Ada routines directly from SETL, typically for graphics. Usually one does not really wish to write new code in the foreign language in such a case—otherwise one does well to write it in the implementation language of some SETL interpreter, and integrate it there. More likely is that one just wants to have an interface to the library in terms natural to SETL. The question then arises: how much effort is it worth to build a "thick", SETL-oriented binding to this library compared to the nearly mechanical generation of a "thin" binding which may require considerable accommodation of the library's needs by the SETL programmer? The answer will depend on how transient the need for that particular library is judged to be.

In 1990, Jack Schwartz had an indefinitely transient need for the facilities of the Macintosh Toolbox, which even then contained over 1000 routines. SETL2 was ported to the Mac, but multiple processes in the standard environment for that platform were not to be available for many years to come. Jack therefore enlisted my help in generating a thin SETL2 interface to the Toolbox. The SETL2 callout interface was exceedingly primitive (all calls had to be routed through a single routine), and Kirk Snyder staunchly refused to allow anyone else access to the SETL2 source code in those days. Partly for these reasons, but mainly due to the fact that there were a great many parameter types, all needing conversions of one kind or another, the C part of this interface was quite bulky, running to some 16,000 lines of mechanically generated code. The SETL programs and other scripts I wrote to generate the interface had the helpful redundancy of Pascal "header" files (source files containing declarations meant to be incorporated into user programs). The C header files did not by themselves convey enough information to distinguish value parameters from result or value-result parameters when an asterisk (indicating a pointer) appeared, but the appearance of var in the corresponding Pascal declaration supplied the needed discrimination in all but a few special cases. Nowadays, good discipline in C headers calls for the use of const to allow programmers to make this distinction at a glance, but this was not common practice in 1990, at least not among the authors of the Macintosh Toolbox.

This was the first of several SETL2 and later SETL interfaces I generated over the years. The Griffin group once even had me generate a SETL2 interface to the X graphics library. Eventually these generators led to what is now a reasonably civilized procedure for customizing my SETL interpreter with thin interfaces to libraries described by C headers. The only such interfaces I have personally found useful to date have been for graphics libraries, such as GLUT/Mesa, which implements a simple event-based windowing system together with an essentially complete realization of OpenGL.

The customization procedure will never be fully automatic, because not all the information pertaining to a library interface is contained in the C header files. Some decisions about the correspondence between C structures and SETL objects have to be made consciously. The goal of customization is principally to extend the SETL library, but except where a very thin interface is acceptable (which is rare), the goal is also to produce a SETL **package** containing "wrapper" routines and other definitions. The work done by the customizer culminates in the production of a Makefile and associated scripts that build files which fit into the structure of the SETL distribution package in such a way that the inclusion of the customization can be selected at configuration time preparatory to compiling and installing the SETL system.

Recently, in a similar vein, a fairly powerful package called SWIG (Simplified Wrapper and Interface Generator) [24] has been developed by Dave Beazley for the purpose of generating interfaces between a number of languages and C/C++ functions (and variables), again based on information found in the kind of C/C++ declarations to be found in header files. Currently, the scripting languages Tcl/Tk [184], Perl [155], and Python [201] are fully supported by SWIG, and there is partial support for Eiffel, Guile, and Java.

Of course, even the best customization procedure or *ad hoc* interface generator cannot ultimately be as good as a properly formalized foreign-language interface. The requisite sublanguage must be able to describe external entities precisely, and to specify how to translate between them and SETL entities. Since low-level languages often deal directly in representations of low-level scalar types and memory layouts, the sublanguage must accommodate these, and observe the restrictions that attend them.

Of all the languages currently in use for systems programming, Ada 95 stands out as the only one able to express such specifications in a way that is simultaneously convenient, comprehensive, and precise. Accordingly, I would make the following proposal.

SETL is badly in need of a respectable implementation, yet it is a small and semantically straightforward language. I believe that the ideal way to write a formal specification for it would be to describe each of its syntactic constructs as expansions into Ada code, and to describe each of its run-time objects using Ada specifications. These two bodies of description obviously dovetail, and the “meta-rules” which are developed to discipline them will form a good basis for describing SETL extensions, including foreign-language interfaces.

The interfacing sublanguage should nominally fit into the style of SETL, but since Ada is well suited to this kind of descriptive role, the SETL forms ought to translate rather directly into Ada.

In fact, it is reasonable to contemplate an “in-line Ada” construct for SETL if Ada is to play such a central definitional role. I have some relevant experience in this regard, as a system I built several years ago called SETL/C++ was a successful though not entirely satisfactory implementation of SETL which used C++ to describe all SETL objects. It had an “in-line C++” feature, which worked perfectly well but was somewhat hard on the eyes. More unsatisfactory was the relative weakness of C++ as a specification language, though I was ultimately able to make the templates etc. do my bidding. But in practical terms, the greatest obstacle to having SETL/C++ take over from the well-worn, C-coded, interpreter-based SETL implementation I still use was the unreliability of C++ compilers, a problem that continues to this day. Ada, on the other hand, is probably the most ideal interface specification language in existence, and GNAT [1] is much more robust than any C++ compiler currently available, so perhaps it is time to let Ada repay SETL for Ada/Ed by using Ada/GNAT to specify and implement the world's first truly robust SETL system.

6.3 Types

There is probably no area in which the advantage of using Ada as a specification language for SETL is more clear than in the area of types. First of all, let me make the general observation that the design of types requires a much higher degree of care and professional experience than virtually any other aspect of program design. This has been evident time and again in the parade of versions of the Java API, the C++ standard library, and even some of the Ada packages that have appeared over the years, especially generics.

Types play a pivotal role in any large system. They are at the core, and much depends on them—they are in many ways the foundation, and I think their centrality makes it fair to consider their design to be in the realm of systems programming rather than applications programming.

SETL, on the other hand, belongs very much in the sphere of applications and, as I have tried to illustrate in this dissertation, is best suited to small programs. Furthermore, its maps and even its tuples serve, however informally, the main purpose of types, which is to package data objects. On their own, maps and tuples do not support formalized abstraction, but the fact that they are values makes them very convenient packets to pass among the routines which do embody an abstraction.

The foregoing helps to explain why the absence of a type system in SETL has been less uncomfortable than it would be in a language like Ada, which aims to support programming in the large at both the systems and the applications level. We see a similar phenomenon in textbooks on algorithms, where the focus is on mechanisms rather than on organizing large bodies of code. Variables are often not even declared, since (for example) seeing $A(i)$ immediately tells us that A is a map, perhaps an array, the presence of and $R.a$ indicates that R is an object with an attribute a, perhaps a record. The genericity in expressions like $A(i)$ extends also to scalars, where in fact it often does not matter what kind of number x is, or even whether x is a number at all. The reader can then take an appropriately abstract view of x until x is seen to be involved in arithmetic or some other expression requiring more specificity.

Still, it is unfortunate that SETL follows the tradition of textbooks to the point of making it difficult to manage a program of significant size, and a non-intrusive type system for SETL would be welcome.

To me, there are two main aspects to how this might be done. One is that type declarations, even when they become available, should remain optional, but it is very important that the compiler and the human reader of a given program unit P come to an understanding about *P* right from the outset: if *P* is marked **strong**, then every variable and formal parameter in *P* must be declared, but if *P* is marked **weak** (the default), then any undeclared variable is implicitly **var**, just as in the current SETL. Programmers writing new code would of course be gently encouraged to declare their program units **strong** to gain the advantages of type checking and possible run-time efficiency improvements, except in cases where this would add more clutter than perspicuity.

The second aspect of non-intrusiveness is that the introduction of a SETL type system should not rob the language of its essential simplicity. This is why I have been at some pains to point out that the design of “core” types is a challenging systems programming activity, and why I began this section by asserting that Ada has advantages as a specification language for SETL. I think the extreme position of introducing new types into a SETL program *only* by slipping into Ada or very Ada-like declarative forms is defensible, and that the ideal approach is not far from that extreme.

However, I feel that SETL should at least acquire some simple forms for defining and extending record types, and for specifying subtypes via Ada-like constraints. These forms should have obvious transliterations into Ada. It is difficult to say more at this stage, except that we should be strictly guided by need in order to avoid inventing myriad declarative forms for SETL of marginal utility. A good starting point is perhaps to add nothing more to SETL than the ability to state that a given type named *T* is described by a piece of Ada code somewhere. Whether *T* is tagged (perhaps all types visible to SETL should be), limited private (perhaps no types visible to SETL should be), etc., and what operations on *T* are described by Ada routines, could all conceivably be specified in pure Ada. If *T* meets the requirements of a SETL type, which could be quite restrictive and stereotyped, doubtless including some uncheckable and even unstated semantic promises, then *T* can be admitted to visibility at the SETL level. There are some operations which apply to all SETL types (like the **type** operator), and undeclared SETL variables will always be able to change their types dynamically, notwithstanding the fact that part of the purpose of a type system will be to help the programmer constrain and group such chameleons appropriately. Dynamic dispatching to routines with fully type-constrained formals should certainly be available. The upshot of the need for these essential features in SETL variables is that they should all derive from some base ADT in Ada that encapsulates the descriptor, or stub, together with the operations that all SETL variables inherit.

There are many issues surrounding types that have not been touched on here, such as (1) the syntax of typed variable declarations in SETL, (2) whether type and variable declarations need to remain segregated from other statements, or can be integrated as in Algol 68 and C++, (3) scopes and interactions with package structure, (4) child packages, (5) generics, (6) inheritance of multiple abstract types, (7) dynamic dispatching based on multiple argument types, (8) communication of typed values between programs, and (9) the possibility of representational “hints” along the lines of the old SETL **repr** clauses.

6.4 String Handling

As remarked in Section 2.14.1, the extensions to SETL's string slicing and subscripting forms to allow selections to be made on the basis of patterns described by regular expressions when **magic** is **true**, and the functions **sub**, **gsub**, **mark**, **gmark**, and **split**, which likewise accept regular expression patterns, have proven to be very useful in their own right, but **sub**/**gsub** and the assigning forms $s(p) := r$ and $s(p_1 \mathinner{..} p_2) := r$ lack the means to refer to matched substrings conveniently in the construction of replacement strings.

The way this is done in the standard Unix 98 editing tools, which effectively support the one-pattern forms (e.g. $s(p) := r$ but not $s(p_1 \mathinner{..} p_2) := r$), is to take an ampersand (&) or backslash-escaped zero (`\0`) in the replacement string to mean the entire matched substring, and to take `\1`, `\2`, etc. to mean substrings that are framed in the pattern by backslash-escaped parentheses, i.e. `\(` and `\)`, where `\`*k* in the replacement string refers to the the *k*th such framed pattern. The backslash-escaped parentheses are not themselves matched against characters in the subject string, but can be nested, and are numbered according to where the left parenthesis begins in the pattern.

SNOBOL [99] and subsequently the MTS [188] editor allow variables to be assigned as a side-effect of pattern matching, by providing syntax that associates variables directly with subpatterns. The "immediate value assignment" form causes these variables to be assigned during the course of the matching process, while "conditional value assignment" defers such assignments until matching is complete (and no assignment will be made if the pattern fails to match the subject string). There is generally no practical reason for preferring conditional value assignment, though efficiency was originally (and probably wrongly) cited. In any case, the resulting values of the variables are available for use in constructing a replacement string or for any other subsequent purpose.

SNOBOL pattern matching is very general. For example, it is fully backtracking, and deferred-value expressions with embedded function calls can be incorporated in patterns. There is a whole suite of predefined patterns and pattern-matching functions. Substring starting positions as well as substrings can be assigned to variables during matching.

Since regular expressions have already been introduced into SETL and are familiar to Unix 98 users, the ideal pattern-matching facility for SETL will integrate their use with some SNOBOL-like ability to incorporate matched substrings in expressions defining replacement strings. Given immediate value assignment into SETL variables, there should be no need for the rather arcane and error-prone ampersand and backslash escape conventions of replacement strings in the regular expression regime of Unix editors, however, since such strings will be able to be constructed by much more powerful means.

Patterns logically form a distinct type, values of which should be produced by certain functions and operators.

For example, the exclamation mark (**!**) might be used as a binary operator that takes a pattern argument *p* on the left and an arbitrary assignment target *t* on the right, and yields a pattern which, when evaluated in the course of matching, assigns to *t* the substring matched by *p*. This is like SNOBOL's immediate value assignment. Such operators should also be overloaded for other types of *p* that serve as primitive patterns, such as **string**, ordered pair of **string**s (consistent with the overloading of **sub**, etc.), and **routine** (a pattern-matching function).

The at-sign (**@**) might be a unary operator that takes an arbitrary assignment target *t* and produces a pattern which matches the null string and assigns to *t* the "cursor position" (string index) of the current location of matching. Again, this corresponds to a SNOBOL primitive operator.

Several primitive functions are also worth borrowing from SNOBOL. For example, **arbno** should take a pattern argument *p* (or **string** etc. that can serve as a pattern) and return a pattern that matches zero or more occurrences of *p*. The functions **pos** and **rpos** should each take an **integer** argument *n* and match the null string if and only if the matching cursor is currently *n* characters from the beginning and end of the string, respectively. Unary overloadings of **any**, **break**, **len**, **notany**, **span**, and **rany** through **rspan**, all of which already appear in SETL as binary functions, are also available for use as pattern-producing functions.

The "unevaluated expressions" of SNOBOL, which are useful in all sorts of pattern-matching situations, could be approximated readily. With no extra syntax, any **tuple** whose first element is a **routine** could serve as a pattern in which subsequent elements, *evaluated prior to matching*, would be passed as arguments to the **routine** when the pattern was encountered during matching. The asterisk ($*$) could be given similar significance as a unary special form when it appears before a global function name f that might or might not be followed by a parenthesized list of arguments: $*f(x_1, x_2, \ldots)$ would be equivalent to [**routine** f, x_1, x_2, $\ldots$] except that it would produce an actual pattern instead of just a tuple that can serve as a pattern. Likewise, $*f$ would be a pattern equivalent to **routine** f or [**routine** f].

The reason for insisting that the arguments intended for f in the foregoing be evaluated when the pattern expression is being constructed rather than when it is being used in matching is that to do otherwise would be to invite dynamic scope violations: patterns can certainly be constructed and yielded by functions. This is also why f must be a *global* function name (as all function names are in the current version of SETL). The asterisk notation could easily be extended to apply to global variables, where it would defer their evaluation until pattern matching time. Nesting, as in $*f(x_1, *g(*y_1, y_2), *x_3)$, is of course perfectly feasible.

Pattern-matching functions yield what they match from the pattern matcher's point of view. They also advance a cursor when they succeed. There are several ways in which user-defined pattern-matching functions could be stereotyped in order to map to this behavior. For example, the subject string and cursor could always be supplied as parameters, or they could be predefined variables that the system pushes and pops when necessary to accommodate pattern matching nested within such a function. The function could indicate failure by yielding **om** and success by yielding the number of characters matched, or perhaps yield the matched substring itself on success. Alternatively, the function could be a predicate, advancing the cursor and yielding **true** on success, but on failure leaving it alone and yielding **false**. “Off-the-shelf” predicates could then be used in patterns to match the null string and allow matching to continue (success), or match nothing and cause the pattern matcher to back up and seek alternatives (failure).

Existing SETL operator symbols are also overloadable for operations such as pattern concatenation and alternation. Alternation could use a new symbol or overload an old one, but concatenation should surely be done with “$+$”, since that is how strings are concatenated. This raises the slight problem that the plus-sign is also used for tuple concatenation, and although some tuples can serve as patterns, not all their concatenations make sense as patterns. For this reason, and because programmers should be encouraged to write code that is free from even the hint of ambiguity, there should be a **pat** operator that converts its argument to a pattern or helpfully complains.

In expressions $s(p) := r$, where s is a string, p is a pattern or equivalent ($s(p_1 \mathinner{\ldotp\ldotp} p_2)$ may be read as $s([p_1,\ p_2])$ under this proposal), and r is a replacement string, it is important to specify something about SETL semantics that was previously left open, namely which of $s(p)$ and r is evaluated first. Clearly, if immediate value assignments in p are to produce values that will be of use in constructing the replacement string r, then $s(p)$ must be evaluated first. Note that s was already specified as the very first part of such forms to be evaluated [181, p. 93], and the last to be assigned. Its value is effectively copied into a temporary t initially. Then matching is performed, and substring replacement is performed on t. Finally the result in t is copied to s. In a complex statement such as

$s(p)(i\,..\,j) := r;$

which is equivalent to

$temp := s(p);$
$temp(i\,..\,j) := r;$
$s(p) := temp;$

the expression $s(p)$ is defined to be evaluated twice, and the programmer must as always beware of *unintended* side-effects. The best policy is to ensure that the side-effects in $s(p)$ consist of nothing more than assignments which can occur either once or twice with equivalent effect. This neither runs afoul of the semantic assumptions nor interferes with the machinations of optimizers.

6.5 Exceptions

The most common use of exceptions is to provide a kind of safety net to deal with errors before they become major problems resulting from the propagation of erroneous values or from wholesale crashes. Some would argue that exceptions should only be used for such purposes, despite the temptation they present to some programmers to use them more as a variant **goto**.

As hinted near the end of Chapter 4, exceptions are not as important in SETL as they might be in other languages. It is certainly true that there are various ways of crashing a SETL program, ranging from systemic errors such as running out of memory to data-oriented errors such as trying to read a non-numeric denotation as a number. The emergence of a SETL type system is likely to introduce more, such as the possibility of constraint violations and other errors.

But all these exigencies can be handled gracefully by isolating the vulnerable code (the kind of code that would normally be placed under the protection of an exception handler) in its own little process attached to the parent through a pump stream. If the child crashes, the parent merely sees an end-of-file condition, which does not crash the parent. The subprocess also offers a level of protection against denial-of-service attacks in which clients try to hold connections open indefinitely (see Section 3.3.1).

However, I make these observations not in order to demonstrate that exception-handling is superfluous, but merely to suggest why its absence has not been a tremendous burden. The fact remains that a good exception-handling facility would be a positive addition to SETL. I will not attempt to lay out a detailed design here, but would like to note that resumption semantics need not be seriously contemplated for SETL. How exactly exceptions are to be identified probably should not be specified until a satisfactory type system has been designed, but the potentially delicate semantics of the initial transfer of control are already equivalently met in the situation where an **expr** block is contained within an expression—a tuple former, for example—and that **expr** block executes a **goto** out of the expression instead of **yield**ing a value.

6.6 Miscellaneous Desiderata

There are numerous minor features which could be added to SETL for the sake of enhancing its already excellent support for Internet data processing, without greatly increasing the complexity of the language. Here we mention a few.

6.6.1 Lexical Nesting

Routines (procedure and operator definitions) cannot be nested in SETL, though procedure nesting is allowed in SETL2. This is of little consequence in small programs from the software engineering point of view, but strictly speaking, any routine *q* that is purely a “helper” for another routine *p* ought to be private to *p*, and the most convenient way of arranging this is to have *q* lexically contained within *p*.

This notion extends to variables and constants as well as routines, and SETL would be improved further by allowing names to be declared locally to control structures, as in Algol 68. The rarely used keyword **begin** should also be imported from Algol 68 if this extension is made, for the sake of doing nothing more than framing a local scope.

Indeed, my own feeling is that SETL would do well to follow the lead of Algol 68 and C++ in allowing declarations to occur anywhere that other statements can occur. A name bound by such a declaration can then be referenced throughout the remainder of the scope. For SETL, a special proviso would have to be made that if a name with no binding applicable anywhere in the current scope occurs, its default declaration is taken to be at the beginning of the innermost enclosing routine (where the main program unit is considered a routine for this purpose). If it is declared in a given scope, the standard rule that says it cannot be referenced in that scope before its point of declaration would apply.

6.6.2 Filename Globbing

All Unix shells are able to create lists of filenames based on patterns that universally include the asterisk (*) as a “wild card” that matches any run of 0 or more characters. The standard Unix 98 shell, and most other Unix shells, also support patterns such as the question mark (?) to match any single character, a bracket-enclosed ([]) run of characters to match any character in that run, and a brace-enclosed ({}) list of strings giving a set of alternatives. For example, if the files foo.c and foo.o are present in the current working directory (see **getwd** and **chdir** in Section 2.6), then the patterns foo.*, foo.?, foo.[co] and foo.{c,o} all stand for the same pair of filenames. Notice that although several characters have special significance in this so-called *globbing* convention, their meaning is different from that which obtains in regular expressions.

In SETL, the most appropriate realization of such a feature would seem to be to introduce a **glob** operator which accepts a string containing a pattern obeying the conventions of the Unix 98 shell and yields a (possibly null) tuple of strings representing filenames that match that pattern. Shells behave similarly, but not identically: if a given pattern does not match any filenames, the standard shell will simply leave the pattern unexpanded, whereas the C shell will issue a diagnostic and abort the process of constructing a list of tokens to form a command. All shells have quoting conventions that allow special characters which are normally expanded to be used as themselves in filenames. In programs, access to such filenames is of course achieved simply by not globbing them.

A glob function has appeared in the Posix [117] specification, and is now part of Unix 98, along with an fnmatch function which tests a single filename to see if it satisfies a given glob-style pattern. Provision of a roughly equivalent SETL primitive would be appropriate when all the main vendors of Unix systems have caught up with these potentially very helpful functions. “Word” expansion in the shell sense would dovetail with this kind of filename expansion, so that abbreviations for user home directory names and other simple expressions that are familiar to shell users could be easily accessible without the need for such convolutions as

```
[fred_home] := split (filter ('bash -c "echo ~fred"'));
```

to obtain a home (login) directory name.

6.6.3 Format-Directed I/O

Another convenience, especially valuable in lower-level programming languages, is format-directed I/O such as that found in Fortran, the C library (standardized by Posix and Unix 98), and Algol 68. COBOL *pictures* are among the most sophisticated formatting features in any popular programming language. The SETL functions **whole**, **fixed**, and **floating** (Section 2.14.2) get their names from Algol 68 routines, which have the interesting property of being one-for-one with symbolic expressions in the format strings of the Algol 68 transput (I/O) system.

The need for format-directed I/O is less in high-level languages than in lower-level languages because it is so easy to build strings and manipulate them as values in the former. Formats also tend to be in rather arcane little sublanguages, which in most cases separate the expression to be output or the variable to be input from the description of its appearance quite widely, making the correspondence difficult to discern. Nevertheless, formats can be quite useful and concise for encoding complex output layouts or dealing with highly structured inputs, particularly as they tend to reflect layouts rather pictorially. My experience with Algol 68, a language of high enough level in its handling of strings and rich enough in its set of I/O primitives to make the use of formats anything but a *sine qua non*, was that for some tasks, they were still to be preferred over long concatenations of string-forming expressions.

For SETL, where there has been some effort in recent years to remain compatible with Unix (a moving but definitely slowing target in this decade), the most natural choice for a format sublanguage would seem to be one which strives to remain close to that of the C-callable printf and scanf series. This has not yet been assessed in serious detail, however.

A related format conversion issue arises for dates and times. The **fdate** primitive described in Section 2.16 can render the number of milliseconds since the beginning of 1970 (UTC) as a date and time in the current time zone or based on UTC, but there is currently no corresponding primitive for taking a formatted date and time apart into constituents in the manner of the Unix 98 strptime routine nor for recombining those parts into a single integer representing time in the manner of mktime.

6.6.4 High-Level Internet Protocols

One of the strengths of the Java API is its support for Internet protocols above the level of UDP and TCP, such as FTP, HTTP, and even (through third-party sources) SMTP, NNTP, and so on. URL "connections" can be opened, and for those which use HTTP, the associated MIME header information can be fetched and set through method calls on the object representing the connection.

In SETL, communication via HTTP is accomplished using a package of SETL routines which must be imported into every SETL program which wants to use them. URLs are probably going to be with us for a long time, and it would be much more convenient and natural to communicate with the entities addressed by URLs through one or more I/O modes such as 'url', 'url-in', and 'url-out', which would be directly supported by **open**. These would be the first modes to participate in the *handling* of the data in streams rather than simply *passing* the data, so some fairly serious design work will be needed here. For example, should MIME headers appear as a map, or be manipulated by a mechanism like **getenv**/**setenv** (Section 2.1), or both? How should non-HTTP protocols such as FTP, which can also be specified with URLs, be treated?

The open-endedness of this problem in fact suggests that the only viable solution will be a modular one, where support for protocols for things like distributed file systems, database systems, and transaction management systems will have to be mediated by add-on modules.

6.7 Beyond the Fringe

There are a number of “features” which might appear at first glance to be desirable in SETL, but which really are not.

6.7.1 Pointers

Perhaps the most obvious example is *pointers*, explicit as in Pascal or Ada (C pointers are so tied to a machine memory model that they permit arithmetic, which is an abomination), or implicit as in Java or SNOBOL. The LISP family would be lost without them. Pointers are so useful in the construction of data structures in various languages that programmers who are used to them may wonder how on earth one is to get along without them. The obvious answer is to use maps. Aliases are, after all, sometimes useful, but as remarked in Sections 1.1 and 5.3.5, they should be avoided except where there is some compelling reason to use them. For situations where aliases are genuinely appropriate, maps are the ideal general reference structure, because they allow keys to be selected on the basis of their semantic content rather than being forced to be opaque nodes or thinly veiled integers, and maps also have the virtue of clearly identifying a context within which each reference (map lookup) occurs—every map is like a separate address space.

Most languages that have pointers include pointers to *variables*. Scheme does not, but the ease with which data can be converted to code at run-time produces a similar effect. Nor does Java—its pointers are all to *objects*—and this at least eliminates one class of alias.

To add pointers to variables in SETL would completely destroy its value semantics. Even to add pointers to objects would be unwise, because maps already best serve the required purpose in a high-level language; there is more to be lost than gained in burdening the SETL programmer with the need to keep track of the distinction between an object and a reference to that object. Neither is it feasible to insist that all user-defined objects have the kind of status they have in Jave, where all accesses begin with a reference. This would make them unacceptably different from SETL's fundamental aggregate objects (sets and tuples), which are strictly values.

6.7.2 Closures and Continuations

Another unwelcome addition to SETL would be *closures* and more generally *continuations* in the Scheme sense. I consider even the creation of "procedure values" by the **routine** pseudo-operator to be a provisional measure that is to be deprecated, despite the fact that it was indispensable in "escaping the event loop" [89] in Section 5.3.8.3 via callbacks. It is of interest that a cautionary note regarding the manipulation of a global variable had to be given with that example.

A closure can preserve local variables from destruction beyond their normal span, so that the state of those variables can be inspected and updated when the closure is later invoked. Closures have some similarity to pointers in that the holder of a closure is effectively in possession of a set of references to variables. Worse, whereas the number of variables to which ordinary procedure values can refer is bounded by the number of global identifiers in a program, there is no such limit for closures. The same pair of lambda expressions, for example, can be used any number of times to produce new pairs of closures, each pair referring to a variable in a different activation record.

Closures are in any case a poor substitute for a more forthright way of bundling data with the means to access and manipulate it, namely objects. Callbacks, too, are better treated in Java/C++ style than being implemented using procedure values or closures. The lightweight multiple inheritance available to any Java object through the repeated use of implements allows it to register for callbacks by any other object that recognizes the appropriate interface simply by defining methods with the names and signatures required by that interface, and identifying itself to that other object by calling the appropriate registration method.

6.7.3 Threads and Fine-Grained Concurrency

This dissertation has argued for the liberal use of processes, and I hope the case study of Chapter 4 has begun to demonstrate their value. These processes enjoy the “high walls of protection” alluded to in Section 5.3.7 around them: they share very few resources, and those which they do share are usually mediated by third parties such as server processes.

By contrast, languages that encourage the use of “threads” (processes sharing variables in the “local” address space) create synchronization concerns for which the programmer must remain constantly vigilant.

I have found Java ideal as a language for introducing concurrency and its attendant problems in three courses so far: a graduate course in Advanced Operating Systems, a senior undergraduate course in Network Programming, and in a junior undergraduate course in Programming Languages, because the elements (thread creation, synchronization, and signalling) are very accessibly packaged in Java, and this allows my students to proceed rapidly to the stage of learning for themselves how difficult concurrent programming is to do correctly. It was rare indeed to find a student program that used the `synchronized` keyword completely appropriately, had no race conditions, and was entirely safe from deadlock.

As with pointers and closures, the issue again is resource sharing by multiple parties. In this case, however, the parties are threads, and instead of only one party at a time making access to common resources, all the parties at once can do so, in an interleaved if not literally concurrent manner.

Apart from disciplined programming, the best defense against the enormous challenges raised by the need to deal with all the possible interactions of simultaneously executing threads is to minimize those interactions. A model that offers the “convenience” of shared access to all the data that is naturally visible to all threads works powerfully against this minimization.

For this reason, even without considering the problems of defining and implementing a SETL system that is correct and yet does a reasonable job of timeslicing and of code optimization, I am very much opposed to the introduction of any kind of threads in SETL that would allow the sharing of global (but process-local) data. Currently, the only objects to which SETL processes share access are external, and in good designs are themselves processes. This is as it should be.

Conversely, threads are not likely to be particularly useful in SETL. Message-passing between processes is at worst a heavier mechanism than strictly necessary in any given situation, and certainly is a sufficient base upon which to implement semaphores, monitors, and so on. In the following example, suggested by Doug Lea's "bounded counter" study [136, pp. 84–102], a monitor for a counter (the counter could as easily be a bounded buffer) is implemented as a process which accepts *increment* requests from "producers" and *decrement* requests from "consumers" through server sockets. Requests are serviced immediately if possible, and the requester is blocked otherwise:

```
-- To run me in Unix:  setl -DMIN=0 -DMAX=2 {me} &

var counter := MIN;      -- MIN effectively #defined on command line
var producers := {};     -- clients awaiting increment
var consumers := {};     -- clients awaiting decrement

inc_sock := open ('0', 'server-socket');     -- listen for producers
dec_sock := open ('0', 'server-socket');     -- listen for consumers

putfile ('inc-port', str port inc_sock);     -- advertise producer port
putfile ('dec-port', str port dec_sock);     -- advertise consumer port
```

```
loop                                    -- cycle indefinitely

  [ready] := select ([{inc_sock, dec_sock}]);

  if inc_sock in ready then     -- producer (increment) request
    fd := accept (inc_sock);
    if fd ≠ om then
      if counter < MAX then     -- satisfy request immediately
        counter +:= 1;
        send_counter_value_to_client (fd);
        notify_any_waiting_consumer;
      else                 -- counter at MAX, defer request
        producers with:= fd;
      end if;
    end if;
  end if;

  if dec_sock in ready then     -- consumer (decrement) request
    fd := accept (dec_sock);
    if fd ≠ om then
      if counter > MIN then     -- satisfy request immediately
        counter −:= 1;
        send_counter_value_to_client (fd);
        notify_any_waiting_producer;
      else                 -- counter at MIN, defer request
        consumers with:= fd;
      end if;
    end if;
  end if;

end loop;

-- Refinements

notify_any_waiting_consumer::
  fd from consumers;        -- pull arbitrary fd from consumers
  if fd ≠ om then
```

```
    counter −:= 1;
    send_counter_value_to_client (fd);
  end if;

notify_any_waiting_producer::
  fd from producers;        -- pull arbitrary fd from producers
  if fd ≠ om then
    counter +:= 1;
    send_counter_value_to_client (fd);
  end if;

-- Subprogram

proc send_counter_value_to_client (fd);
  printa (fd, counter);     -- tell client the new counter value
  close (fd);
end proc;
```

This program can be exercised conveniently by running as many instances of the following programs as desired. For simplicity, they assume they are being run on the local machine in the same directory as the monitor, but because the communication uses full-fledged TCP sockets, they can easily be run anywhere if the monitor's actual host name and port numbers are first substituted in the **open** calls:

```
-- Producer program (requests one increment)
fd := open ('localhost:' + getfile 'inc-port', 'socket');
print (getline fd);

-- Consumer program (requests one decrement)
fd := open ('localhost:' + getfile 'dec-port', 'socket');
print (getline fd);
```

6.8 SETL Implementations

As was remarked in Chapter 1, there has never really been a fully satisfactory SETL implementation, although my own has at least the advantages of being actively maintained, released under the GNU Public License, written in portable C, blessed with a very fast translator, and packaged in a way which makes heavy use of autoconf to allow it to be configured and adapted easily to a wide variety of Unix-based operating systems.

However, its interpreter reflects a much greater interest in modularity than speed. When a new version of the language which includes a respectable type system has been sufficiently well defined, I think it would be worthwhile to write an entirely new SETL compiler and run-time system in Ada 95. If the language is defined in terms of Ada expansions, and the compiler is designed around that definition, it will be possible to take advantage of the GNAT [1] system. Deeper code transformations like those performed by APTS [33, 161, 126, 162], in the presence of a serious type system, might permit the kind of thorough optimization usually seen only in the implementations of lower-level languages.

Another advantage of tying SETL's definition and implementations to Ada is that Ada's run-time semantics are readily expressed in term of the Java Virtual Machine (JVM) [129]. This would pave the way to writing browser "applets" in SETL.

6.9 Comparison with Other Languages

The main thing that distinguishes SETL from other high-level languages is its syntax. Its roots in mathematical notation, developed and refined by and for people long before computers arrived on the scene, lend it the ability to express a wide variety of computational processes naturally, elegantly, and above all readably.

6.9.1 Perl

Perl [182, 205, 155] is perhaps the most commonly used "high-level" language for data processing. It sprang from Larry Wall's frustration with the insufficiency of awk for these purposes, and has grown into a baroque monstrosity through endless patching. The syntax is arcane and turgid (practically every variable reference has to start with a special symbol), and the semantics are just as bad: all undeclared variables are global, and the system interface is so implementation-dependent as to make a mockery of the claims of portability usually made for Perl. Worse, the interface to Unix is so thin that, for example, select cannot be used with the normal buffered I/O streams but only with raw file descriptors, and the semantics of signals depend on which strain of Unix is hosting the Perl implementation. The latter means that re-establishing the signal handler may well suffer from the historical System V race condition that led to the BSD-style and eventually Posix signal handling definitions. Unix routines that are not re-entrant have the same potential for causing disaster if called from signal handlers in Perl as they would in C, and to top it all off, signals will be fielded asynchronously regardless of the state of the garbage-collecting memory manager, leaving the documentation no choice but to advise programmers to "do as little as possible" in signal handlers to minimize the probability of disaster—no guarantee can be made in this regard, no matter how simple the handler is kept.

SETL, of course, strictly shields the high-level programmer from such nonsense, by providing a high-level interface to **select**, by permitting only synchronous access to signals, and in general by keeping all issues of portability internal to the run-time system rather than exposing them at an inappropriate level. Moreover SETL, unlike Perl, provides bidirectional pipe streams (called pumps in SETL), a powerful inter-process communication facility whose usefulness is amply demonstrated by the WEBeye case study of Chapter 4.

Perl is often cited for conciseness of expression, but I have never found it to beat SETL on that score either. Indeed, the opposite is often true. Here, for example, is a TCP client program derived rather directly from the one in the "llama" book [182, p. 224], where it is described as a "simple client". This version is modified to be functionally identical to the one-line SETL program in Section 3.1.1, and is also similar to the Perl sample program that can be found on the perlipc manual page on most Unix systems:

```
$port = 13;
$them = ’galt.cs.nyu.edu’;

use Socket;

$sockaddr = ’S n a4 x8’;
```

```
($name, $aliases, $proto) = getprotobyname('tcp');

($name, $aliases, $type, $len, $thisaddr) = gethostbyname('localhost');
($name, $aliases, $type, $len, $thataddr) = gethostbyname($them);

$this = pack($sockaddr, AF_INET, 0, $thisaddr);
$that = pack($sockaddr, AF_INET, $port, $thataddr);

socket(S, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
connect(S, $that) || die "connect: $!";

print <S>;
```

Kernighan and Pike [133] code the classic Markov chain algorithm in C, Java, C++, Awk, and Perl to compare them on a typical small data processing task. The program sizes range from 150 lines of source code for the C version down to 18 lines for the Perl version, reproduced here:

```
# markov.pl:  markov chain algorithm for 2-word prefixes

$MAXGEN = 10000;
$NONWORD = "\n";
$w1 = $w2 = $NONWORD;           # initial state
while (<>) {                    # read each line of input
  foreach (split) {
    push(@{$statetab{$w1}{$w2}}, $_);
    ($w1, $w2) = ($w2, $_);     # multiple assignment
  }
}
push(@{$statetab{$w1}{$w2}}, $NONWORD);     # add tail
```

```
$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
   $suf = $statetab{$w1}{$w2};   # array reference
   $r = int(rand @$suf);         # @$suf is number of elems
   exit if (($t = $suf->[$r]) eq $NONWORD);
   print "$t\n";
   ($w1, $w2) = ($w2, $t);       # advance chain
}
```

The authors comment that “the Perl and Awk programs are short compared to the three earlier versions, but they are harder to adapt to handle prefixes that are not exactly two words” [133, p. 80]—indeed, in the versions written in the lower-level languages, the prefix size was governed by a named integer constant. Perl is very much on home turf here, and in fact this is exactly the sort of program Perl programmers love to write. The following SETL version is only slightly shorter than the above Perl program on which it is modeled, but is more general (the number of prefix words is controlled by a constant) and more direct (for example, the tuple of words recorded in each *statetab* entry is built up by tuple concatenation rather than with some mysterious push operator). To me, it also looks like a real program instead of a wild festival of glyphs:

```
-- Markov chain demo

const maxgen = 10000;
const npfx = 2;              -- number of prefix words
const nonword = '\n';

statetab := {};
words := npfx * [nonword];            -- initial state
for word in split (getfile stdin) loop
```

```
    $statetab(words)$ +:= [$word$];
    $words$ := $words$(2 .. ) + [$word$];
  end loop;
  $statetab(words)$ +:= [$nonword$];        -- add tail

  $words$ := $npfx$ * [$nonword$];
  for $i$ in [1 .. $maxgen$] loop
    if ($word$ := random $statetab(words)$) = $nonword$ then stop; end if;
    print($word$);
    $words$ := $words$(2 .. ) + [$word$];
  end loop;
```

6.9.2 Icon

Icon [97, 96] is an interesting high-level language for programming in the small. It has no type declarations, and its variable declarations and scope rules are similar to those of SETL. It represents something of an extreme in programming language design in that its expression evaluation mechanism is fully backtracking over *generators* which *suspend* and produce a value when first encountered or when asked to *resume* by being backed into. Unless it is abandoned before it is exhausted, a generator must ultimately *fail*. Together with some rules which limit backtracking in appropriate contexts, this provides a convenient basis for control flow, because the assignment of expression results into variables typically occurs at the same time as successful evaluation, eliminating the need for a distinct Boolean type.

Icon's pervasive backtracking is a generalization of the backtracking that was used in SNOBOL string pattern matching. I have used both Icon and SNOBOL's near-identical twin SPITBOL extensively, and although the tight interweaving of control flow with expression evaluation in Icon sometimes produces welcome effects, such as the fact that i < j < k accomplishes what in most languages would have to be written as (i < j) and (j < k), the extra "liveness" of the ubiquitous generators somehow does not turn out to be so often useful as it does to require vigilance. The backtracking that worked so well for pattern matching in SNOBOL is perhaps at its best in that limited domain. Stranger still, the syntactic improvements to pattern matching that were made never turned out to be quite as comfortable as the original SNOBOL forms.

Recently, Icon has proven itself to be particularly well suited to graphics programming [98], and of course it has always been strong on string manipulation. My feeling is that Icon is a close second to SETL for data processing. Its syntax is fairly elegant, though not as close to that of mathematics as SETL's (in particular, it doesn't have set and tuple formers), and is rather heavy with operator forms—for example, "not equals" is spelled ~=, ~==, or ~===, depending on whether numbers, strings, or general objects such as lists are being compared. Icon also lacks SETL's value semantics, as is evident in the fact that the outcome of === or ~=== depends on whether the objects are *identical,* which for lists means both operands refer to the same list, but for scalars simply means that they have the same value.

Icon allows record definitions which, like procedures, are global. The fieldnames are not typed, and may be addressed by name or by position. Records are lumped in with lists, sets, and tables as the aggregates of values collectively called *structures* in Icon terminology. They all have pointer semantics, and a copy primitive performs "one-level" copying. It is tempting to adopt a record model for SETL that is as simple as Icon's, perhaps borrowing its cavalier attitude towards field typing and reference though not its pointer semantics. I think this should wait, however, until an Ada-based type system such as that suggested in Section 6.3 has at least been investigated far enough to ensure that no unfortunate incompatibilities will be introduced.

6.9.3 Functional Languages

LISP and derivatives such as Scheme [128], and the purely functional languages such as ML [146] and Haskell [197], are of more theoretical than practical interest for data processing, though Haskell's *list comprehensions* are very similar to SETL's tuple formers, and its *lazy evaluation* admits of concise expressions that can represent infinite lists much more directly than functions can. Haskell's strong typing combined with the ability to dispatch to a routine of the correct signature based on generalized *pattern matching* lend it great potential for clarity and elegance. Haskell does not seem to have found its way into the world of data processing to any significant extent. I do not know why, but can only surmise that the lack of assignment is disconcerting to many. In the case of servers, it is not clear that tail recursion is the most natural way to express an infinite main loop, nor whether state information such as a map of current client records is best represented as a succession of values, each map (say) being constructed from the previous map and passed as a parameter to the next round of recursion. On the other hand, if one assumes that the optimizer and garbage collector in a Haskell implementation are able to recognize the opportunity for conversion of tail recursion to iteration and the possibility of suppressing copy proliferation, this approach does offer the significant advantage of avoiding the hazards associated with the updating of variables in place.

6.9.4 Python

Python [201] is another language that is close in level to SETL, and is deliberately oriented towards what I have been calling data processing in this dissertation. It descends from ABC [193], a language which Guido van Rossum, the designer and implementer of Python, helped design in the 1980s. Python has a number of odd and in some cases counterintuitive aspects, such as its scoping rules [147]. One of the interesting features it inherits from ABC is the use of indentation as the only syntactic mechanism for control-structure grouping. Python suffers from the regrettable pleonasm of simultaneously offering *lists*, which are tuples with pointer semantics, and *tuples*, which are tuples with value semantics. Tuple denotations are marked by parentheses, and the problem of how to denote a singleton is unfortunately solved by writing, e.g., “(x,)”.

The typical assertions of conciseness for a high-level language are made on behalf of Python [200], but it does not have anything approaching SETL’s set and tuple formers. As a consequence, even a “showpiece” subroutine such as the following one [199] by van Rossum can be written more concisely and readably in SETL:

```
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if not graph.has_key(start):
        return None
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath: return newpath
    return None
```

In SETL, the equivalent function may be written with an existential predicate in place of the for-loop:

```
proc find_path (graph, first, last);
    if first = last then
        return [first];
    end if;
    if exists next in graph{first} |
            (path := find_path (graph lessf first, next, last)) ≠ om then
        return [first] + path;
    end if;
    return om;
end find_path;
```

This assumes a multi-map *graph*, but can be modified to accommodate an adjacency list by changing “*graph*{*first*}” to “*graph*(*first*) ? {}”. The difference in languages becomes even more apparent in the following example, again by van Rossum:

```
def find_all_paths(graph, start, end, path=[ ]):
    path = path + [start]
    if start == end:
        return [path]
    if not graph.has_key(start):
        return [ ]
```

```
paths = [ ]
for node in graph[start]:
    if node not in path:
        newpaths = find_all_paths(graph, node, end, path)
        for newpath in newpaths:
            paths.append(newpath)
return paths
```

In SETL, this becomes

```
proc find_all_paths (graph, first, last);
   if first = last then
      return {[first]};
   end if;
   return {[first] + path :
               path in +/{find_all_paths (graph lessf first, next, last) :
                              next in graph{first}}};
end find_all_paths;
```

where the structure of the result, a set of paths that are tuples, is more obvious and at the same time more natural than the list of lists returned by the Python function. And of course in SETL, if we are content to have paths represented as subgraphs and are more interested in a concise definition than in micro-managing the optimization process, we can simply write

```
proc find_all_paths (graph, first, last);
   return {s in pow graph | is_path (s, first, last)};
end proc;
```

where *is_path* might be defined as

```
**proc** $is\_path\ (graph, first, last)$;
   $vertices := $ **domain** $graph$ + **range** $graph$;
   **return** $\#graph = \#vertices - 1$ **and**
     **domain** $graph = vertices$ **less** $last$ **and**
     **range** $graph = vertices$ **less** $first$;
**end proc**;
```

which tests for *graph* being a tree whose domain is all vertices except *last* and whose range is all vertices except *first*. Finally, *find_path* is easily implemented as **arb** applied to the result of *find_all_paths*.

6.9.5 Rexx

Rexx [49], originally a shell language that began to supersede the primitive EXEC job control language in the CMS component of IBM's VM/370 operating system during the 1980s, has been used for many years now as a general-purpose high-level language, and has been ported to a wide variety of systems. It retains many of the characteristics of a shell programming language. For example, any commands that are not recognized as designating built-in or user-defined operations are passed to the environmental command interpreter, such as CMS or an editing environment such as that operating system's standard XEDIT.

Although all values manipulated by a Rexx program are strings (even arithmetic is defined only on strings representing numbers), Rexx has both *simple* variable names such as x and *compound* ones such as x.y. The latter serve as a general associative device, because the y in x.y can be a number, a simple variable name, or a symbol that is neither a number nor a variable name. Moreover, if `y` is the name of an uninitialized variable, its default value is '`Y`', i.e., its own name in uppercase.

This gives considerable power to a language that is fundamentally very simple. Its parser has to be present at run time, because any string can be treated as an expression to be evaluated. Its scopes are unrepentantly dynamic. Rexx is a good "glue" language, suitable for high-level control, but it lacks both the mathematical syntax and data structuring capabilities of SETL. Although its compound variable names allow it to be used directly for data processing, these shortcomings and the usual pitfalls of a highly dynamic language are likely to be felt more acutely as program size increases.

6.9.6 Java

There are many more languages which could be compared with SETL as high-level data processing languages, but let us conclude by reflecting on one that is really of intermediate level but has gained much currency in recent years: Java. This is a language which took advantage of the widespread popularity of C++ to make an immediate appeal to programmers. It is backed by huge financial resources, is well documented and widely implemented, and (perhaps most important of all) has APIs for a great variety of application domains defined for it.

The brilliance of Java as a language is really its reductionism. Its syntax is very thin—it doesn't have the operator declarations of C++, Algol 68, or SETL, and certainly nothing close to SETL's set and tuple formers. Its object inheritance facilities are also very streamlined. For this, however, it probably deserves more praise than criticism. A class can only be derived from one other *class*, but it can also be derived from any number of *interfaces*, which are purely abstract descriptors with no data of their own. This eliminates the troublesome questions that arise in general multiple inheritance, such as what happens when a base class with data occurs more than once in the hierarchy of ancestors, yet the Java model still allows an object to play multiple roles, such as identifying itself as a party interested in several different kinds of event.

Everything is an object in Java, except for a few built-in types of scalar value. This is a less welcome aspect of its reductionism, as it encourages the creation of aliases instead of independent values (cf. Sections 5.3.5 and 6.7.1). Its threads encourage the *asynchronous* sharing of variables, which is worse, as outlined in Section 6.7.3.

Java's approach to the handling of exceptions may seem a little severe in its tendency to force programmers to consider errors they might not be interested in, but this is certainly to be preferred to the approach which doesn't allow errors to be caught at all, especially in long-running programs intended for public use.

The best thing to do with Java is probably to follow the lead of the Intermetrics corporation in adapting a better language, Ada 95 in that company's case, to Java's magnificent suite of APIs, and use the Java language as the mid-level, symbolic form of what is undeniably a highly portable machine language, Java Byte Code. The maintainers of Rexx have also taken this approach. JPython [115] goes even farther, being an entirely new implementation of Python written in Java and targeted to the Java runtime environment. Each of these systems takes advantage of and reflects Java package structure in some way, and SETL would do well to follow suit.

6.10 Summary

Among programming languages, SETL stands apart as the only one to have been designed to take advantage of the syntactic tradition of abstract mathematics. Schwartz's desire to free programmers from their preoccupation with machine-level concerns, and his recognition that people had already evolved notations which served as an aid to mathematical reasoning, combined to produce a language which to this day is unparalleled in its ability to express the most essential programming constructions concisely *and* naturally. It should perhaps come as no surprise that SETL gracefully handles the relatively mundane generalizations required in typical data processing. SETL is an elegant language, with clean semantics and a refreshing degree of orthogonality. The importance of these attributes to those who must deal daily with programs is profound. It is easy to take pride in one's work when the results tend to reward revisiting: a virtuous cycle of readability, robustness, and maintainability ensues.

SETL is a language which deserves to be taken seriously. I have tried to show in this dissertation how it excels in a realm remote from the algorithmic showpieces of 1970, how it can be really useful in the process-intensive age of the Internet, and what general principles and patterns seem to work best in this context. There is much work still to be done, but it is perhaps not too optimistic to hope that SETL's slow start will one day prove to have been a good start.

Appendix A

WEBeye Source Code

Following are the source code listings for the WEBeye Box described in Chapter 4. They appear in alphabetical order by filename, and are cross-referenced according to their roles and relationships. The script precursors vc-master.cgi and vc-jmaster.cgi are not strictly part of the Box, but call into it. Likewise, the programs vc-go.setl, vc-quit.setl, vc-restart.setl, vc-check.setl, and vc-cron.setl are outside the Box but deal with aspects of its administration.

The typographic conventions here, as throughout the text of the dissertation, reflect the result of automated preprocessing, which uses fonts, subscripts, and open quotes for clarity. The SETL sources from which these listings are derived are encoded in a subset of the "printable" ASCII characters. They obey the character set restrictions of the SETL textbook [181], and can be compiled under the default case-insensitive stropping convention of my current SETL implementation [19].

A.1 vc-admin.setl

Textually #included by:

vc-cron.setl (Section A.9)
vc-go.setl (Section A.18)
vc-quit.setl (Section A.35)
vc-restart.setl (Section A.37)
vc-toplev.setl (Section A.42)

Source code:

```
--  Management services, principally relating to the creation,
--  interpretation, and removal of mutual exclusion (mutex) locks.
--
--  Use:  #include me after you define yhwh and my_lock.

proc msg (s);  -- log directly on stdout (cf. msg in vc-msg.setl)
 spew (yhwh + ' : ' + s);
end proc;

proc spew (s);  -- log a timestamped message
 print (fdate(tod), ':', s);
 flush (stdout);
end proc;

proc commence;  -- acquire mutex or abend immediately
 var text, t, rc;
 make_symlink;
 if last_error ≠ no_error then
  msg ('Cannot obtain lock file ' + my_lock + ' - ' + last_error);
  if (text := readlink my_lock) ≠ om then
   msg ('Another instance of ' + yhwh + ' may be running.');
   msg ('Currently, ' + render_lock text + '.');
   msg ('Please check processes and do a ' + str ('rm ' + my_lock) +
                                         ' if necessary.');
   if yhwh = 'vc-toplev.setl' then
    msg ("You can use 'vc-quit' to make sure the Box is stopped,");
    msg ("and then 'vc-check' to check for old processes.");
    msg ("Finally, 'vc-restart' will clear out " + str my_lock);
```

```
        msg ("and manage the log files for you.");
      end if;
    end if;
    rc := 1;
    msg ('Exiting with status = ' + str rc + '.');
    stop rc;
  end if;
end proc;

proc make_symlink;
  clear_error;
  symlink (yhwh + ' (pid ' + str pid + ') started at ' + str tod,
           my_lock);
end proc;

op render_lock (text);
  return render_link (my_lock, text);
end op;

proc render_link (name, text);
  var t;
  return name + ' -> ' + text +
         if (t := extract_timestamp text) $\neq$ om
         then ' [' + fdate(t) + ']'
         else ''
         end if;
end proc;

-- Extract the last "field" in text as a timestamp if possible:
op extract_timestamp (text);
  var t;
  t := split (text);
  t := t(#t);
  if t('[1-9][0-9]*') = t then
    t := val t;
    if t $\leq$ tod + 10*365*24*60*60*1000 then
      return t;
    end if;
```

```
 end if;
 return om;
end op;

-- Extract the pid embedded in text if possible:
op extract_pid (text);
 var p;
 if (p := text('\\(pid ' .. '\\)')) ≠ om then
  p := p(6 .. #p−1);
  if p('[1-9][0-9]*') = p then
   p := val p;
   -- This is well beyond the current range of Unix pid numbers:
   if p < 2**31 then
    return p;
   end if;
  end if;
 end if;
 return om;
end op;

proc finis (rc);  -- release mutex and yield status code rc to system
 unlink (my_lock);
 stop rc;
end proc;
```

A.2 vc-allowed.setl

Textually #included by:
 vc-do.setl (Section A.11)
 vc-event.setl (Section A.12)
 vc-image.setl (Section A.20)
 vc-toplev.setl (Section A.42)

Source code:

```
-- Test whether the peer at peer_address fd is in the set of
-- "allowed" hosts (usually meaning allowed to use the server
-- that has called allowed):

proc allowed (fd, hosts(*)); -- rudimentary host-based security
  var ok_ips, afd, ip;
  [ok_ips] := hosts;
  if ok_ips = om then
    afd := open ('vc-allowed.conf', 'r');
    if afd = om then
      ok_ips := {} +/ {aliases ip : ip in {'localhost', '127.0.0.1',
                                            hostname,   hostaddr}};
    else
      reada (afd, ok_ips);
      close (afd);
    end if;
  end if;
  ip := peer_address fd;
  return ok_ips * aliases ip ≠ {};
end proc;

op aliases (ip);
  return ip_names (ip) + ip_addresses (ip);
end op;
```

A.3 vc-autoinit.setl

Called by parent program:
  vc-send.setl (Section A.38)
Calls child programs:
  vc-comdev.setl (Section A.7)
  vc-comport.setl (Section A.8)

Source code:

```
const yhwh = 'vc-autoinit.setl';

-- Try to provoke the Canon into auto-initializing

printa (stderr, yhwh, 'begins');

com_dev  :=     filter ('exec setl vc-comdev.setl');
com_port := val filter ('exec setl vc-comport.setl');

msr := 6;  -- modem status register (MSR)
cts_bit := 2**4;  -- "clear to send" (CTS) bit

fd := fileno open ('/dev/port', 'direct');

-- Obtain initial CTS value
seek (fd, com_port + msr);
msr_val := abs getc (fd);
cts_val := sign (msr_val bit_and cts_bit);
printa (stderr, 'CTS at entry to', yhwh, ':', cts_val);

-- Try to provoke an auto-init by sending a bad length byte
putfile (com_dev, '\xff');

start_clock := clock;

-- Delay 300 ms to give it plenty of chance to take effect
select (om, 300);

seek (fd, com_port + msr);
```

```
msr_val := abs getc (fd);
cts_val := sign (msr_val bit_and cts_bit);
printa (stderr, 'CTS after 300 ms :', cts_val);

-- Watch for CTS to come back up.

loop doing
  delta := clock − start_clock;
while delta < 7500 do
  select (om, 100);  -- delay 100 ms
  seek (fd, com_port + msr);
  msr_val := abs getc (fd);
  if sign (msr_val bit_and cts_bit) = 1 then
    printa (stderr, 'CTS up after', delta, 'ms');
    goto done;
  end if;
end loop;
printa (stderr, 'CTS not up after', delta, 'ms');

done:
printa (stderr, yhwh, 'ends');
```

A.4 vc-camera.setl

Service provided:
  camera
Called by parent program:
  vc-toplev.setl (Section A.42)
Calls child program:
  vc-ptz.setl (Section A.33)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-provide.setl (Section A.32)

Source code:

```
const yhwh = 'vc-camera.setl';

-- This is a multiplexing server "front end" to the program
-- vc-ptz.setl which provides a high-level command
-- interface, designed for the convenience of telnet
-- clients and other programs, to the Canon VC-C3 pan/tilt/zoom
-- camera controller.

const vc_ptz = 'exec setl vc-ptz.setl';

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const camera_fd = fileno provide_service ('camera');

var clients := {};

loop

  [ready] := select ([{sigterm_fd, camera_fd} + domain clients]);

  if sigterm_fd in ready then
    msg (yhwh + ' (' + str pid + ') caught SIGTERM');
    quit_gracefully;
  end if;
  for client = clients(pump_fd) | pump_fd in ready loop
    msg (clients(pump_fd).name + ' done');
    close (pump_fd);
    clients(pump_fd) := om;
  end loop;

  if camera_fd in ready then
    fd := accept (camera_fd);
    if fd ≠ om then
      name := getname fd;
      msg (name + ' accepted');
      pump_fd := open (vc_ptz + ' -- ' + str fd, 'pump');
      close (fd);
      if pump_fd ≠ om then
        client := {};
        client.name := name;
        clients(pump_fd) := client;
      else
        msg ('failed to start "' + vc_ptz + '" for ' + name);
      end if;
    end if;
  end if;

end loop;

proc quit_gracefully;
  exit_gracefully ([[str filename pump_fd+' for client '+client.name,
                         pump_fd] : client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.5 vc-check.setl

Source code:

```
-- Report on Box processes (though this program is not part of the Box)

const pid_dir = ‘vc-pid’;

const ps = filter(‘/bin/ps alxwwj’);
const lines = split(ps,‘\n’);
const header = split(lines(1));
const pstup = [split(line) : line in lines(2 .. ) | #line > 0];
const pid_index = index(‘PID’);
const ppid_index = index(‘PPID’);
const command_index = index(‘COMMAND’);
const pid_map = {[tup(ppid_index),  -- main parent->child map
         tup(pid_index)] : tup in pstup};
const finder = {[tup(pid_index), i] : tup = pstup(i)};

for name in split (filter (‘/bin/echo ’+pid_dir+‘/*’)) |
    name notin {‘’, pid_dir+‘/*’, pid_dir+‘/vc-toplev’} loop
  root := getfile name;
  id := val root;
  if not is_integer id or id ≤ 0 or id ≥ 2**31 then
    printa (stderr, ‘Process id in file ’+str name+‘ is invalid!’);
  else
    name(pid_dir+‘/’) := ‘’;
    if pexists id and root in range pid_map then
      printa (stderr, ‘Process tree for "’+name+‘":’);
      t := [format pstup(finder(root))];
      if root in domain pid_map then
        t +:= doit (root, 0);
      end if;
      n := 0 max/[#line : line in t];
      for j in [1,4 .. n] loop
        prot := false;
        for i in [#t,#t−1 .. 1] loop
          c := t(i)(j);
          if c = '|' then
            if not prot and (mark (t(i), '\\\\'))(1) > j then
              t(i)(j) := ' ';
            end if;
          elseif c = '\\' then
            prot := true;
          else
            prot := false;
          end if;
        end for i;
      end for j;
      for line in t loop
        line(' $') := '';  -- fanatic
        printa (stderr, line);
      end for line;
    else
      printa (stderr, 'Process '+root+' for "'+name+'" is gone.');
    end if;
  end if;
end for name;

name := pid_dir+'/vc-toplev';
if (root := getfile name) ≠ om then
  id := val root;
  if not is_integer id or id ≤ 0 or id ≥ 2**31 then
    printa (stderr, 'Process id in file '+str name+' is invalid!');
  else
    name(pid_dir+'/') := '';
    if pexists id then
      assert root in range pid_map;
      line := format pstup(finder(root));
      printa (stderr, 'Primordial process for "'+name+'":');
      printa (stderr, line);
    else
      printa (stderr, 'Process '+root+' for "'+name+'" is gone.');
    end if;
  end if;
else
```

```
  name(pid_dir+'/') := '';
  printa (stderr, 'There is no record of the process for "'+name+'".');
end if;

if (filter ('uname'))('Linux') ≠ om then
  printa (stderr, 'Try also "ps fxj" to check for orphaned processes.');
else
  printa (stderr, 'Try also "ps xl" to check for orphaned processes.');
end if;


proc index (what);
  assert exists field = header(i) | field = what;
  return i;
end proc;

proc doit (parent, k);
  return [ ] +/ [ [k*'|  '+'\\_ '+format pstup(finder(child))] +
               doit (child, k+1) : child in pid_map{parent} ];
end proc;

op format (tup);
  return tup(pid_index) + ' = '
        +/[field+' ' : field in tup(command_index..)];
end op;
```

A.6 vc-clear.setl

Called by parent program:
  vc-send.setl (Section A.38)
Calls child programs:
  vc-comdev.setl (Section A.7)
  vc-comport.setl (Section A.8)

Source code:

```
const yhwh = 'vc-clear.setl';

-- "Clear" the Canon

printa (stderr, yhwh, 'begins');

com_dev := filter ('exec setl vc-comdev.setl');
com_port := val filter ('exec setl vc-comport.setl');
rts_lo_time := val (command_line(1) ? '150');  -- ms
mcr := 4;  -- modem control register
rts_bit := 2**1;
fd := fileno open ('/dev/port', 'direct');

seek (fd, com_port + mcr);
mcr_val := abs getc (fd);
mcr_val bit_and:= bit_not rts_bit;
seek (fd, com_port + mcr);
putc (fd, char mcr_val);  -- put RTS low
select (om, rts_lo_time);  -- delay rts_lo_time ms

seek (fd, com_port + mcr);
mcr_val := abs getc (fd);
mcr_val bit_or:= rts_bit;
seek (fd, com_port + mcr);
putc (fd, char mcr_val);  -- put RTS hi
select (om, 150);  -- give the Canon 150 ms to catch its breath

printa (stderr, yhwh, 'ends');
```

A.7 vc-comdev.setl

Called by parent programs:
vc-autoinit.setl (Section A.3)
vc-clear.setl (Section A.6)
vc-init.setl (Section A.21)
vc-input.setl (Section A.22)
vc-send.setl (Section A.38)

Source code:

```
-- Echo either the contents of the file containing the configured
-- serial device name, or a default name.

const conf_file = 'vc-comdev.conf';
[com_dev] := split (getfile conf_file ? '/dev/ttyS0');
putchar (com_dev);
```

A.8 vc-comport.setl

Called by parent programs:
vc-autoinit.setl (Section A.3)
vc-clear.setl (Section A.6)

Source code:

```
-- Echo either the contents of the file containing the configured
-- serial port address, or a default port address.

const conf_file = 'vc-comport.conf';
[com_port] := split (getfile conf_file ? '16#3f8');
putchar (com_port);
```

A.9 vc-cron.setl

Calls child program:
  vc-restart.setl (Section A.37)
Textually #includes:
  vc-admin.setl (Section A.1)

Source code:

```
const yhwh = 'vc-cron.setl';

-- This is supposed to be run every minute from the user's crontab,
-- using an entry like this:
--
--  * * * * * cd fred; setl vc-cron.setl
--
-- This program basically tries to detect and correct any problems with
-- the Box that is supposed to be permanently running.  It tries to do
-- so without going crazy (like the Sorcerer's Apprentice) in the
-- attempt.
--
-- If and when it decides that the Box is down or malfunctioning, and
-- that it is time to try to correct the problem, it will try to clean
-- up (by removing apparently stale locks and processes) and restart
-- the Box.

const my_lock    = 'vc-cronlock';   -- lock file (mutex)
const vc_cronlog = 'vc-cronlog';    -- log of how I exited
const vc_lock    = 'vc-lock';       -- Box's lock file
const vc_link    = 'vc-link.html';  -- link to pseudo- or other document
const vc_prefix  = 'CGI|';          -- pseudo-document convention
const vc_health  = 'vc-tcp/health';  -- has Box's health check host:port
const vc_camera  = 'vc-tcp/camera';  -- has Box's "command" host:port
const vc_going      = 'vc-going';       -- vc-go's lock file
const vc_quitting   = 'vc-quitting';    -- vc-quit's lock file
const vc_restarting = 'vc-restarting';  -- vc-restart's lock file
const vc_rescount   = 'vc-rescount';  -- restarts since last "success"
const vc_restart_cmd = 'exec setl vc-restart.setl';  -- restart command
```

```
var box_locktext;


-- Try to acquire the lock named by my_lock, if appropriate.
--
-- If another recently started instance of this program is already
-- running, exit quietly and immediately.  Otherwise, try to correct
-- the problem with the lock and exit more noisily, in the hope that
-- the cleanup effort (and any action the administrator subsequently
-- takes) will make it easy to get when the program is run again soon
-- (like in another minute):
--
make_symlink;
if (make_symlink_result := last_error) ≠ no_error then
 if (locktext := readlink my_lock) ≠ om then
  stamp := extract_timestamp locktext ? 0;
  if stamp < tod − 15∗60∗1000 then
   -- The lock is at least 15 minutes old, or bogus.
   msg (‘Lock file ’ + render_lock locktext +
      ‘ is more than ’ + str ((tod − stamp) div 60000) +
      ‘ minutes old.’);
   -- Try to find and blow away the old instantiation of this
   -- program, remove the lock, and exit.
   if (oldpid := extract_pid locktext) ≠ om then
    if pexists oldpid then
     kill_process (oldpid);
    else
     msg (‘However, process ’ + str oldpid +
        ‘ appears to be no longer active.’);
    end if;
   else
    msg (‘Cannot find pid in ’ + str locktext + ‘.’);
   end if;
   msg (‘Removing ’ + my_lock + ‘ and exiting with status = 1.’);
   cron_exit (1, render_lock locktext + ‘ old or bogus’);
  else
   -- Active instance, if any, is not considered too old.
   if (oldpid := extract_pid locktext) ≠ om then
```

```
        if pexists oldpid then
          -- Bow out gracefully.
          cron_log (0, 'instance ' + str oldpid + ' still active.');
          stop;
        else
          msg ('Lock file ' + render_lock locktext +
               ' indicates a young and active process, but ' +
               str oldpid + ' appears to be no longer active.');
          msg ('Removing ' + my_lock + ' and exiting with status = 1.');
          cron_exit (1, 'instance ' + str oldpid +
                                  ' disappeared mysteriously');
        end if;
      else
        msg ('Cannot find pid in ' + str locktext + '.');
        msg ('Removing ' + my_lock + ' and exiting with status = 1.');
        cron_exit (1, 'no pid found in ' + render_lock locktext);
      end if;
    end if;
  else
    msg ('Cannot create lock file ' + my_lock + ' for reason ' +
         str make_symlink_result + ', but cannot read it as a ' +
         'symlink either, for reason ' + str last_error + '.');
    msg ('Attempting unlink of ' + my_lock + '.');
    clear_error;
    unlink (my_lock);
    if last_error ≠ no_error then
      msg ('Unlink attempt failed for reason ' + str last_error + '.');
    elseif lexists my_lock then
      msg ('Unlink appeared to succeed, but ' + my_lock +
         ' still exists.');
    else
      msg ('Unlink apparently successful.');
    end if;
    msg ('Exiting with status = 2.');
    cron_exit (2, 'problem with access to ' + my_lock);
  end if;
end if;
-- We have now acquired the mutex lock named in my_lock.
-- Now see if the Box seems to be up and healthy.

-- The Box lock file is supposed to be a symbolic link to a string of
-- information (not a real file).  Try to read that information:
if (box_locktext := readlink vc_lock) = om then
  failure ('Cannot read ' + vc_lock + ' as a symbolic link');
end if;

-- Look for the Box's process id embedded in the link
if (boxpid := extract_pid box_locktext) = om then
  failure ('Could not find pid in ' + str box_locktext);
end if;

-- Check whether the process thus identified really exists
if not pexists boxpid then
  failure ('Process ' + str boxpid + ' indicated in lock file ' +
          render_link (vc_lock, box_locktext) + ' has disappeared');
end if;

-- Wait to see pseudo-document indicating that Box is up and available
--- magic-constants file?  vc-limits.setl?
interval := 100;  -- ms
limit := 100000;  -- ms
loop for ms in {interval,2*interval .. limit} doing
  flag := (content := getfile vc_link) ≠ om and
        match (content, vc_prefix) = vc_prefix;
while not flag do
 select (om, interval);
end loop;
if not flag then
  failure (box() + ' failed to reach "running" state');
end if;

-- Try to open the Box's health-check service
fd := open_box_service (vc_health, 'health check');

-- Exercise the health check
```

```
if (line := getline fd) ≠ 'ok' then
  failure (box() + ' failed health check, reason = ' + str line);
end if;
close (fd);  fd := om;

-- Try to open Box's camera-control command service
fd := open_box_service (vc_camera, 'camera control command');

-- Absorb its opening niceties
while split (getline fd ? '.') ≠ ['.'] loop
  pass;
end loop;

-- Tell it to ensure consistency between hardware and software state
printa (fd, 'check');
close (fd);  fd := om;

-- Zero the restart counter, unlink lock, and exit "successfully"
putfile (vc_rescount, '0');
cron_exit (0, 'OK');


-- Fancy Box identifier
proc box();
  return 'Box {' + render_link (vc_lock, box_locktext) + '}';
end proc;

proc open_box_service (vc_hpfile, what);
  -- Look for the designated service of the Box
  if (hp := getfile vc_hpfile) = om then
    failure (box() + ' failed to create file ' + str vc_hpfile);
  end if;
  -- Try to open hp, the host:port of the designated service
  if (fd := open (hp, 'socket')) = om then
    failure (box() + ' not listening on ' + what + ' port ' + hp);
  end if;
  return fd;
end proc;
```

```
-- This routine is called when the Box appears to be down or
-- malfunctioning, and attempts a restart if conditions conduce:

proc failure (message);

  -- Update count of restart attempts made since the last time a
  -- properly functioning Box was detected:
  [raw] := split (getfile vc_rescount ? '0');
  if raw('[1-9][0-9]*') ≠ raw and raw ≠ '0' then
    msg (vc_rescount + ' file corrupted - contains ' + str raw +
         ' instead of a number - treating as 0');
    n := 0;
  else
    n := val raw;
  end if;
  n +:= 1;
  putfile (vc_rescount, str n);

  -- In case of recurring failures, this ratchets the restart attempts
  -- back in powers of 2 up to a maximum interval of 64 units (which is
  -- 64 minutes if this program is run every minute by cron):
  if n mod 64 = 0 or exists i in [0 .. 5] | 2**i = n then

    -- Try to restart the Box.
    -- First try to make sure there aren't any active instances of
    -- vc-go, vc-quit, or vc-restart:
    should_run_restart := true;
    loop for lockfile in [vc_going, vc_quitting, vc_restarting]
         while should_run_restart do
      if (locktext := readlink lockfile) ≠ om then
        stamp := extract_timestamp locktext ? 0;
        if stamp < tod - 10*60*1000 then
          -- Lock file more than 10 minutes old, or bogus
          msg ('Removing lock file ' + render_lock locktext);
          unlink (lockfile);
          if (stale_pid := extract_pid locktext) ≠ om then
            if pexists stale_pid then
              kill_process (stale_pid);
            end if;
          else
            msg ('Could not find pid in ' + str locktext);
          end if;
        else
          -- Lock file still quite young.  Skip this restart
          -- opportunity - another will come along presently:
          should_run_restart := false;
        end if;
      end if;
    end loop;
    if should_run_restart then
      -- Attempt restart, let the world know about it, and exit:
      msg (message + ' - attempting to restart Box');
      system (vc_restart_cmd);
      cron_exit (1, message + ' - performed ' + str vc_restart_cmd);
    end if;

  end if;

  -- Log the failure, unlink the lock, and exit without spamming
  -- stderr or attempting a restart
  cron_exit (1, message);

end proc;


proc kill_process (process_id);
  msg ('Killing ' + str process_id + ' ...');
  kill (process_id);  -- send TERM signal
  select (om, 333);  -- give TERM a chance
  kill (process_id, 'KILL');  -- send KILL signal to make sure
end proc;


-- The log file named in vc_cronlog should only be used for this
-- purpose; other spam will be spewed on stderr and mailed to
-- the user by cron:
proc cron_log (rc, message);
  fd := open (vc_cronlog, 'a');
  printa (fd, fdate(tod), ': pid', pid, ': exit', rc, ':', message);
  close (fd);
end proc;

proc cron_exit (rc, message);
  cron_log (rc, message);
  finis (rc);
end proc;


#include "vc-admin.setl"
```

A.10 vc-decode.setl

Textually #included by:
- vc-do.setl (Section A.11)
- vc-event.setl (Section A.12)
- vc-model.setl (Section A.27)
- vc-ptz.setl (Section A.33)

Source code:

```
-- Operator to decode a low-level message from the Canon

op decode (frame);

 const camera_head = 1,
       pan_tilter = 5,
              ccu = 8;

 r := {};

 if not is_string frame or #frame = 0 then
   r.kind := 'erroneous';
   r.frame := frame;
   r.detail := if is_string frame then
                 'Zero-length frame'
               else
                 if frame = om then 'Frame is OM'
                 else 'Frame is of type ' + type frame
                 end
               end if;
   return r;
 end if;

 if #frame = 1 then  -- one of my specials
   r.kind := 'special';
   r.frame := frame;
   case frame(1) of
   ('*'):  r.detail := 'Inter-character timeout';
   ('@'):  r.detail := 'Ack timeout in multiple command sequence';
```

```
  ('+'):  r.detail := 'Checksum error';
  ('!'):  r.detail := 'Timeout waiting for ack/nak or response';
  end case;
  return r;
end if;

r.frame := hex frame;

fid := abs frame(1);  -- Frame ID

device := fid bit_and 16#0f;

case device of
(camera_head):  r.device := 'Camera Head';
(pan_tilter):   r.device := 'Pan/Tilter';
(ccu):          r.device := 'CCU';
else            r.device := 'unknown device (0x'+hex char device+')';
end case;

if (fid bit_and 16#80) = 0 then
  -- command frame
  cid := abs frame(2);  -- Command ID
  cmd := cid bit_and 16#1f;  -- Command Type bits
  r.cmd := 'unknown (0x'+hex char cmd+')';  -- default
  if (cid bit_and 16#80) = 0 then
    -- Host → Canon message
    r.kind := 'request';
    case device of
    (camera_head):
      case cmd of
      (16#04):  r.cmd := 'Status';
      (16#10):  r.cmd := 'Focus';
      (16#12):  r.cmd := 'Zoom';
      (16#14):  r.cmd := 'Exposure';
      end case;
    (pan_tilter):
      case cmd of
      (16#04):  r.cmd := 'Status';
      (16#10): r.cmd := 'Set-up';
      (16#11): r.cmd := 'Home position';
      (16#12): r.cmd := 'Pan/tilt';
      (16#17): r.cmd := 'Remote controller';
      (16#19): r.cmd := 'LED';
      end case;
     (ccu):
      case cmd of
      (16#01): r.cmd := 'Software reset';
      (16#04): r.cmd := 'Status';
      (16#10): r.cmd := 'White balance';
      (16#11): r.cmd := 'Fade';
      (16#12): r.cmd := 'Mute';
      (16#17): r.cmd := 'Control mode select';
      (16#18): r.cmd := 'Preset';
      end case;
     end case;
  else
     -- Canon → Host message
     if (cid bit_and 16#20) = 0 then
      -- Response
      if (cid bit_and 16#40) = 0 then
      r.kind := 'positive response';
      else
      r.kind := 'negative response';
      end if;
      case device of
      (camera_head):
      case cmd of
      (16#04): r.cmd := 'Status';
      (16#10): r.cmd := 'Focus';
      (16#12): r.cmd := 'Zoom';
      (16#14): r.cmd := 'Exposure';
      end case;
      (pan_tilter):
      case cmd of
      (16#04): r.cmd := 'Status';
      (16#10): r.cmd := 'Set-up';
```

```
      (16#11):  r.cmd := 'Home position';
      (16#12):  r.cmd := 'Pan/tilt';
      (16#17):  r.cmd := 'Remote controller';
      (16#19):  r.cmd := 'LED';
      end case;
    (ccu):
      case cmd of
      (16#01):  r.cmd := 'Software reset';
      (16#04):  r.cmd := 'Status';
      (16#10):  r.cmd := 'White balance';
      (16#11):  r.cmd := 'Fade';
      (16#12):  r.cmd := 'Mute';
      (16#17):  r.cmd := 'Control mode select';
      (16#18):  r.cmd := 'Preset';
      end case;
    end case;
  else
    -- Notification
    r.kind := 'notification';
    case device of
    (camera_head):
      case cmd of
      (16#03):  r.cmd := 'Error';
      (16#11):  r.cmd := 'Focus limit';
      (16#13):  r.cmd := 'Zoom limit';
      (16#18):  r.cmd := 'Button operation';
      end case;
    (pan_tilter):
      case cmd of
      (16#03):  r.cmd := 'Error';
      (16#16):  r.cmd := 'Limit';
      (16#18):  r.cmd := 'Remote controller';
      (16#1b):  r.cmd := 'Power';
      end case;
    (ccu):
      case cmd of
      (16#03):  r.cmd := 'Error';
      end case;
```

```
        end case;
        if r.cmd = 'Error' then
          if #frame ≥ 3 then
            error_type := abs frame(3);
            if #frame ≥ 4 then
              error_cause := abs frame(4);
            else
              error_cause := om;
            end if;
            [r.error_type,
             r.error_cause] := decode_error (error_type,
                                             error_cause);
          end if;
        end if r.cmd;
      end if (cid bit_and 16#20);  -- response vs notification
    end if (cid bit_and 16#80);  -- "command" direction

  else
    -- ack/nak frame
    ack_nak_id := abs frame(2);
    if ack_nak_id = 0 then
      r.kind := 'ack';
      r.detail := 'Received';
    else
      r.kind := 'nak';
      r.detail := decode_nak_cause (ack_nak_id);
    end if;

  end if (fid bit_and 16#80);  -- "command" vs ack/nak

  return r;

end op decode;

proc decode_error (error_type, error_cause);
  r_error_type := decode_error_type (error_type);
  r_error_cause := om;
  if error_cause ≠ om then
  case error_type of
  (16#01):  -- Camera communication error
   -- The “cause” here is the raw UART(?) status byte
   r_error_cause := ‘Status register 0x’ + hex char error_cause;
  (16#03, 16#04):  -- Pan/tilter or RS-232C comm. error
   r_error_cause := decode_comm_error_cause (error_cause);
   if error_cause = 16#00 and #frame ≥ 5 then  -- NAK
    r_error_cause +:= ‘ (’+decode_nak_cause (abs frame(5))+‘)’;
   end if;
  end case;
 end if;
 return [r_error_type, r_error_cause];
end proc;

proc decode_error_type (error_type);
 return case error_type of
 (16#00):  ‘Camera cable is disconnected’,
 (16#01):  ‘Camera communication error’,
 (16#03):  ‘Pan/tilter communication error’,
 (16#04):  ‘RS-232C communication error’,
 (16#10):  ‘Command for unconnected device’,
 (16#11):  ‘Undefined command’,
 (16#12):  ‘Undefined parameter’,
 (16#13):  ‘Status error (received command)’,
 (16#14):  ‘Status error (received parameter)’,
 (16#16):  ‘Timeout’,
 (16#18):  ‘White balance correction error’,
 (16#19):  ‘Pan/tilt, zoom, focus limit’
 else      ‘unknown error type (0x’ + hex char error_type + ‘)’
 end case;
end proc;

proc decode_comm_error_cause (cause);
 return case cause of
 (16#00):  ‘NAK received’,
 (16#01):  ‘ACK receive timeout’,
 (16#02):  ‘Checksum error’,
 (16#03):  ‘Pan/tilter is busy’,
 (16#04):  'Fatal error',
 (16#05):  'Sequence error',
 (16#06):  'Length error',
 (16#07):  'Buffer is busy'
 else      'unknown comm. error cause code (0x' + hex char cause + ')'
 end case;
end proc;

proc decode_nak_cause (cause);
 return case cause of
 (16#01):  'Buffer busy',
 (16#02):  'Length error',
 (16#03):  'Sequence error',
 (16#04):  'Communication error',
 (16#10):  'Checksum error'
 else      'unknown NAK cause code (0x' + hex char cause + ')'
 end case;
end proc;
```

A.11 vc-do.setl

Services provided:
  do, used by local clients:
    vc-httpd.setl (Section A.19)
    vc-jumper.setl (Section A.25)
    vc-mouse.setl (Section A.28)
    vc-mover.setl (Section A.29)
    vc-ptz.setl (Section A.33)
    vc-zoomer.setl (Section A.43)
  notice, used by local clients:
    vc-evjump.setl (Section A.13)
    vc-evzoom.setl (Section A.14)
    vc-ptz.setl (Section A.33)
Client of service:
  event (vc-event.setl, Section A.12)
Called by parent program:
  vc-toplev.setl (Section A.42)
Calls child program:
  vc-model.setl (Section A.27)
Textually #includes:
  vc-allowed.setl (Section A.2)
  vc-decode.setl (Section A.10)
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)

Source code:

```
const yhwh = 'vc-do.setl';

-- This program provides the do and notice services.
--
-- The do service is a server interface to model, a pumping
-- co-process which maintains a high-level model of the videocamera
-- control state and supports "mid-level" commands (requests already
-- reduced to SETL maps) to alter that state.  Besides routing such
-- commands from clients into the model subprocess, do also
-- implements a queuing policy which allows every client that cannot be
-- satisfied immediately to have a command pending, and also prevents
-- further commands from that client from being queued until the
-- pending one has been performed.
--
-- The notice service distributes “mid-level” events to all
-- interested clients.  These originate as low-level events generated
-- by the event service and as responses to parameter-changing
-- commands issued to the model pump.

const model_pump = ‘exec setl vc-model.setl’;

const sigterm_fd = open (‘SIGTERM’, ‘signal’);  -- catch TERM signals

-- Performer of mid-level commands:
const model_fd = fileno open (model_pump, ‘pump’);

-- Generator of low-level events:
const event_fd = fileno obtain_service (‘event’);

const do_server_fd = fileno provide_service (‘do’);
const notice_server_fd = fileno provide_service (‘notice’);

var notice_clients := {};  -- map from client fd to client record
var do_clients := {};  -- map from client fd to client record
var do_queue := [];  -- queue of fd’s of do clients awaiting service
var do_pending := false;  -- true when we await a reply from model

open (‘SIGPIPE’, ‘ignore’);  -- as in when we write to closed observers

loop
  nonwaiting := domain do_clients − {do_fd : do_fd in do_queue};
  pool := if do_pending then {model_fd} else {} end if
        + {sigterm_fd, event_fd, do_server_fd, notice_server_fd}
        + nonwaiting;
  [ready] := select ([pool]);
```

```
if sigterm_fd in ready then
  msg (yhwh + ' (' + str pid + ') caught SIGTERM');
  quit_gracefully;
end if;

for do_fd in ready * nonwaiting loop
  -- New request from a do client.
  reada (do_fd, request);
  if eof then
    do_clients(do_fd) := om;
    close (do_fd);
  else
    do_client := do_clients(do_fd);
    do_client.request := request;
    do_clients(do_fd) := do_client;
    do_queue with:= do_fd;
  end if;
end loop;

if do_pending and model_fd in ready then
  reada (model_fd, model_response);
  if eof then
    msg ('EOF from '+str model_pump+' - quitting');
    quit_gracefully;
  end if;
  -- These notices can be created by the model pump to let us alert
  -- all the observers to parameter changes and special events such
  -- as initialization:
  for message in model_response.notices ? [ ] loop
    tell_observers (message);  -- tell notice clients
  end loop;
  do_fd fromb do_queue;
  do_client := do_clients(do_fd);
  request := do_client.request;
  if request.name = 'Get' then
    writea (do_fd, model_response.value);
  else
    printa (do_fd);  -- a blank line to say the command has been done
  end if;
  flush (do_fd);
  do_pending := false;
end if;

if event_fd in ready then
  reada (event_fd, frame);
  if eof then
    msg ('EOF from '+filename event_fd+' - quitting');
    quit_gracefully;
  else
    message := decode frame;
    tell_observers (message);  -- tell notice clients
  end if;
end if;

if do_server_fd in ready then
  do_fd := accept (do_server_fd);
  if do_fd ≠ om then
    name := getname do_fd;
    if allowed (do_fd) then
      do_client := {};
      do_client.name := name;
      do_clients(do_fd) := do_client;
    else
      close (do_fd);
      msg (name+' denied access to "do" service');
    end if;
  end if;
end if;

if notice_server_fd in ready then
  notice_fd := accept (notice_server_fd);
  if notice_fd ≠ om then
    name := getname notice_fd;
    if allowed (notice_fd) then
      notice_client := {};
      notice_client.name := name;
      notice_clients(notice_fd) := notice_client;
    else
      close (notice_fd);
      msg (name+' denied access to "notice" service');
    end if;
  end if;
end if;

if #do_queue > 0 and not do_pending then
  do_fd := do_queue(1);
  do_client := do_clients(do_fd);
  request := do_client.request;
  writea (model_fd, request);
  flush (model_fd);
  do_pending := true;
end if;

end loop;


proc tell_observers (message);
for notice_client = notice_clients(notice_fd) loop
  clear_error;
  writea (notice_fd, message);
  flush (notice_fd);  -- eventually causes EPIPE if client closed
  if last_error ≠ no_error then
    close (notice_fd);
    notice_clients(notice_fd) := om;
  end if;
end loop;
end proc tell_observers;

proc quit_gracefully;
exit_gracefully ([[str model_pump, model_fd]]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
```

```
#include "vc-getname.setl"
#include "vc-allowed.setl"
#include "vc-decode.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.12 vc-event.setl

Services provided:
  event, used by local client vc-do.setl (Section A.11)
  notify, used by local client vc-seq.setl (Section A.39)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-allowed.setl (Section A.2)
  vc-decode.setl (Section A.10)
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-provide.setl (Section A.32)

Source code:

```
const yhwh = 'vc-event.setl';

-- Low-level event notification distributor

var producers, consumers;

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals

-- Producers use this "notify" port to supply events to us:
const in_server_fd = fileno provide_service ('notify');

-- Consumers receive events on this "event" port:
const out_server_fd = fileno provide_service ('event');

open ('SIGPIPE', 'ignore');  -- as when we write to closed consumers

producers := {};
consumers := {};

loop do

  [ready] := select ([{sigterm_fd, in_server_fd, out_server_fd} +
                                    domain producers]);

 if sigterm_fd in ready then
   msg (yhwh + ' (' + str pid + ') caught SIGTERM');
   quit_gracefully;
 end if;

 for producer = producers(fd) | fd in ready loop
   reada (fd, event);
   if eof then
     close (fd);
     producers(fd) := om;
   else
     tell_consumers (event);
   end if;
 end loop;

 if in_server_fd in ready then
   fd := accept (in_server_fd);
   if fd ≠ om then
     name := getname fd;
     if allowed (fd) then
       producer := {};
       producer.name := name;
       producers(fd) := producer;
     else
       msg (name+' not allowed as event producer');
       close (fd);
     end if;
   end if;
 end if;

 if out_server_fd in ready then
   fd := accept (out_server_fd);
   if fd ≠ om then
     name := getname fd;
     if allowed (fd) then
       consumer := {};
```

```
    consumer.name := name;
    consumers(fd) := consumer;
   else
    msg (name+‘ not allowed as event consumer’);
    close (fd);
   end if;
  end if;
 end if;

end loop;

proc tell_consumers (event);
 for consumer = consumers(fd) loop
  clear_error;
  writea (fd, event);
  flush (fd);  -- should eventually cause EPIPE if client closed
  if last_error ≠ no_error then
   msg (‘consumer ’+consumer.name+‘ done’);
   close (fd);
   consumers(fd) := om;
  end if;
 end loop;
end proc;

proc quit_gracefully;
 -- Degenerate, since we currently have no pump- or pipe-attached child
 exit_gracefully ([ ]);
end proc;

#include “vc-provide.setl”
#include “vc-getname.setl”
#include “vc-allowed.setl”
#include “vc-decode.setl”
#include “vc-exit.setl”
#include “vc-msg.setl”
```

A.13 vc-evjump.setl

Service provided:
  evjump
Client of service:
  notice (vc-do.setl, Section A.11)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-javent.setl (Section A.23)

Source code:

```
const yhwh = 'vc-evjump.setl';

-- Serve motion events in a very simple form designed for Java clients

-- The name of the service we provide:
#define event_type 'evjump'

-- Initial tokens of the event type we look for:
const of_interest = ['Move', 'To'];

-- The numbers represent pan (azimuth) and tilt (elevation):
const n_parms = 2;

#include "vc-javent.setl"
```

A.14 vc-evzoom.setl

Service provided:
  evzoom
Client of service:
  notice (vc-do.setl, Section A.11)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-javent.setl (Section A.23)

Source code:

```
const yhwh = 'vc-evzoom.setl';

-- Serve zoom events in a very simple form designed for Java clients

-- The name of the service we provide:
#define event_type 'evzoom'

-- Initial tokens of the event type we look for:
const of_interest = ['Zoom'];

-- The number represents the zoom factor:
const n_parms = 1;

#include "vc-javent.setl"
```

A.15 vc-exit.setl

Textually #included by:

- vc-camera.setl (Section A.4)
- vc-do.setl (Section A.11)
- vc-event.setl (Section A.12)
- vc-giver.setl (Section A.17)
- vc-httpd.setl (Section A.19)
- vc-image.setl (Section A.20)
- vc-javent.setl (Section A.23)
- vc-model.setl (Section A.27)
- vc-mouse.setl (Section A.28)
- vc-ptz.setl (Section A.33)
- vc-push.setl (Section A.34)
- vc-recv.setl (Section A.36)
- vc-seq.setl (Section A.39)
- vc-simpler.setl (Section A.40)
- vc-snap.setl (Section A.41)

Source code:

```
-- Send SIGTERM signals to all processes listed in name_fd_pairs
-- asking them to quit, and then quit the current process via stop.
-- First we ask politely, and if that doesn't work, we force the issue.


proc exit_gracefully (name_fd_pairs);
 var sigchld_fd, name, fd, id, wid, moniker, ready, line, exited;
 sigchld_fd := open ('SIGCHLD', 'signal');  -- catch CHLD signals
 exited := {};
 for [name, fd] in name_fd_pairs loop
  if fd ≠ om then
   id := pid (fd);
   moniker := name + ' (pid ' + str id + ')';
   msg ('TERMinating ' + moniker);
   clear_error;
   kill (id);  -- send SIGTERM
   if last_error = no_error then
    -- The subprocess either existed or was already a zombie.
    -- First try to clear it quickly.  If the process existed
    -- when the signal was sent, we will receive a SIGCHLD as
    -- soon as it manages to exit, and the following select
    -- will then unblock.  If the process was already a zombie,
    -- this select will only hold up the show for 50 ms, as it
    -- will if the process exists but does not exit that quickly:
    [ready] := select ([{sigchld_fd}], 50);
    if sigchld_fd in ready then
      line := getline sigchld_fd;
    end if;
    -- If the process has exited (and is therefore now a zombie
    -- unless it has already been waited for), it can be cleared
    -- from the process table and added to our exited set:
    while (wid := wait (false)) > 0 loop
      exited with:= wid;
    end loop;
    if id in exited then
      -- The subprocess has exited, so we can safely call close
      -- on its file descriptor without blocking, and carry on.
      close (fd);
      continue for [name, fd] in name_fd_pairs loop;  -- next!
    end if;
    if sigchld_fd notin ready then
      -- We have no indication of the subprocess having exited yet.
      -- Give it a bit more time.  We didn't want to start out
      -- this way, because if the subprocess had already been a
      -- zombie, we would have waited the full 1414 ms:
      [ready] := select ([{sigchld_fd}], 1414);
      if sigchld_fd in ready then
        line := getline sigchld_fd;
      end if;
      while (wid := wait (false)) > 0 loop
        exited with:= wid;
      end loop;
      if id in exited then
        close (fd);
        continue for [name, fd] in name_fd_pairs loop;  -- next!
      end if;
```

```
        end if;
        -- The subprocess seems reluctant to exit.  Hit it harder.
        msg ('KILLing ' + moniker);
        kill (id, 'KILL');  -- send SIGKILL
        [ready] := select ([{sigchld_fd}], 100);
        if sigchld_fd in ready then
          line := getline sigchld_fd;
        end if;
        while (wid := wait (false)) > 0 loop
          exited with:= wid;
        end loop;
        if id in exited then
          close (fd);
          continue for [name, fd] in name_fd_pairs loop;  -- next!
        end if;
        -- This probably indicates a bug in the host OS
        msg ('*** ' + moniker + ' failed to exit on KILL signal');
      else
        msg ('*** ' + moniker + ' not found');
      end if;
    end if;
  end loop;
  close (sigchld_fd);
  msg (yhwh + ' (' + str pid + ') exiting');
  stop;
end proc;
```

A.16 vc-getname.setl

Textually #included by:

vc-camera.setl (Section A.4)
vc-do.setl (Section A.11)
vc-event.setl (Section A.12)
vc-giver.setl (Section A.17)
vc-httpd.setl (Section A.19)
vc-image.setl (Section A.20)
vc-javent.setl (Section A.23)
vc-mouse.setl (Section A.28)
vc-ptz.setl (Section A.33)
vc-push.setl (Section A.34)
vc-simpler.setl (Section A.40)
vc-snap.setl (Section A.41)
vc-toplev.setl (Section A.42)

Source code:

```
-- Return the best available identifier of a client, including the
-- IP name if possible (otherwise just the IP address) followed by
-- a colon and the remote (usually ephemeral) port number.

op getname (fd);
  return (peer_name fd ? peer_address fd ? 'ANONYMOUS') + ':' +
          str peer_port fd;
end op;
```

A.17 vc-giver.setl

Service provided:
  giver
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)

Source code:

```
const yhwh = 'vc-giver.setl';

-- This makes up and gives out a stream of URLs for a stream of
-- JPEGs.  It should go away when when the more civilized Java API
-- (1.2) is widely available.

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const giver_fd = fileno provide_service ('giver');

var clients := {};

loop

 [ready] := select ([{sigterm_fd, giver_fd} + domain clients]);

 if sigterm_fd in ready then
  msg (yhwh + ' (' + str pid + ') caught SIGTERM');
  quit_gracefully;
 end if;

 for client = clients(pump_fd) | pump_fd in ready loop
  done_client (pump_fd);
 end loop;
 if giver_fd in ready then
  fd := accept (giver_fd);
  if fd ≠ om then
   name := getname fd;
   msg (name+' accepted');
   pump_fd := pump();
   if pump_fd = −1 then
    -- child
    [serv_name, serv_port] := find_service ('snap');
    serv_name('localhost') := hostaddr;  -- for public consumption
    pfx := 'http://'+serv_name+':'+str serv_port+'/?seq=';
    i := 0;
    while (line := getline fd) ≠ om and
          split (line) = ['JPEG'] loop
     i +:= 1;
     name := str i;
     printa (fd, pfx+name);
    end loop;
    stop;
   end if;
   close (fd);  -- child hangs onto this
   client := {};
   client.name := name;
   clients(pump_fd) := client;
  end if;
 end if;

end loop;

proc done_client (pump_fd);
 msg (clients(pump_fd).name + ' done');
 close (pump_fd);
 clients(pump_fd) := om;
end proc done_client;

proc quit_gracefully;
 exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                                        client = clients(pump_fd)]);
```

```
end proc;

#include “vc-provide.setl”
#include “vc-obtain.setl”
#include “vc-getname.setl”
#include “vc-exit.setl”
#include “vc-msg.setl”
```

A.18 vc-go.setl

Called by parent program:
    vc-restart.setl (Section A.37)
Calls child program:
    vc-toplev.setl (Section A.42)
Textually #includes:
    vc-admin.setl (Section A.1)

Source code:

```
const yhwh = 'vc-go.setl';

-- Start the Box

const my_lock = 'vc-going';  -- lock file (mutex)
const vc_lock = 'vc-lock';   -- Box's lock file
const vc_log  = 'vc-log';    -- Box's log file
const vc_cmd  = 'exec setl vc-toplev.setl';  -- Box's top-level program
const vc_link = 'vc-link.html';  -- eventually points to pseudo-document
const vc_prefix = 'CGI|';        -- pseudo-document "magic" convention
const vc_health = 'vc-tcp/health';  -- host:port for Box's health check

var box_locktext;


commence;  -- acquire mutex or exit abnormally right away

interactive := system ('tty -s') = 0;


if (box_locktext := readlink vc_lock) ≠ om then
 if interactive then
  printa (stderr, "The Box's lock file still exists:");
  printa (stderr, render_link (vc_lock, box_locktext));
  putc (stderr, 'Shall I forcibly remove it? [n/y] ');
  if yes ('no') then
   clear_error;
   unlink (vc_lock);
```

```
      if last_error = no_error then
        printa (stderr, 'Lock file ' + vc_lock + ' removed.');
      else
        printa (stderr, 'Error "' + last_error + '" trying to remove ' +
                        vc_lock + '.');
        msg ('Abending.');
        finis (1);
      end if;
    else
      printa (stderr, 'Lock file ' + vc_lock + ' not removed.');
      msg ('Exiting.');
      finis (0);
    end if;
  else
    msg ("Error - Box's lock file still exists:");
    msg (render_link (vc_lock, box_locktext));
    msg ('Abending without starting the Box.');
    finis (1);
  end if;
end if;

if fexists vc_log then
  if interactive then
    printa (stderr, 'The log file ' + vc_log + ' already exists.');
    putc (stderr, 'Go ahead and clobber it? [n/y] ');
    if yes ('no') then
      printa (stderr, 'Okay, ' + str vc_cmd + ' will clobber ' +
                              vc_log + '.');
    else
      printa (stderr, 'Okay, log file left intact.');
      printa (stderr, "Try 'vc-restart' in order to save it and then");
      printa (stderr, 'start the Box again.');
      msg ('Exiting.');
      finis (0);
    end if;
  else
    msg ('Warning - ' + str vc_cmd + ' will clobber ' +
                vc_log + '.');
  end if;
end if;

full_cmd := vc_cmd + ' > ' + vc_log + ' 2>&1';
msg ('Starting ' + str full_cmd + ' in the background ...');
system (full_cmd + ' &');

-- Wait for Box's lock to reappear
loop for ms in {10,20 .. 250} while not lexists vc_lock do
  select (om, ms);  -- 10 + 20 + ... + 250 ms = 3.25 sec
end loop;
if (box_locktext := readlink vc_lock) = om then
  failure ('Box failed to create lock file ' + vc_lock + '.');
end if;
if interactive then
  printa (stderr, 'Box has succeeded in creating its lock file:');
  printa (stderr, render_link (vc_lock, box_locktext));
end if;

-- Wait to see pseudo-document indicating that Box is up and available
--- magic-constants file?  vc-limits.setl?
interval := 100;  -- ms
limit := 100000;  -- ms
if interactive then
  printa (stderr, 'Box initializing ... please wait (up to ' +
                 str (limit/1000) + ' sec.)');
end if;
loop for ms in {interval,2*interval .. limit} doing
  flag := (content := getfile vc_link) ≠ om and
         match (content, vc_prefix) = vc_prefix;
while not flag do
  select (om, interval);
  if interactive then
    putc (stderr, fixed (ms/1000, 6, 1) + ' sec.\r');
  end if;
end loop;
if interactive then
  printa (stderr);
```

```
end if;
if not flag then
  failure (box() + ' failed to reach "running" state.');
end if;

-- Try to exercise its health check
if (hp := getfile vc_health) = om then
  failure (box() + ' failed to create file ' + str vc_health + '.');
end if;
-- hp is now the host:port designation of the health-check service
if (fd := open (hp, 'socket')) = om then
  failure (box() + ' failed to open health-check service at ' + hp + '.');
end if;
if (line := getline fd) ≠ 'ok' then
  failure (box() + ' failed health check, reason = ' + str line);
end if;
close (fd);

msg (box() + ' appears to be up and running normally.');
msg ('Done.');
if interactive then
  printa (stderr, 'You may wish to "tail -f ' + vc_log +
                '" for a while.');
end if;
finis (0);  -- release mutex and exit normally


proc yes (default);
  [ans] := split (getline stdin);
  ans ?:= default;
  return to_lower ans in {'y', 'yes'};
end proc;

-- Fancy Box identifier
proc box();
  return 'Box {' + render_link (vc_lock, box_locktext) + '}';
end proc;
```

```
proc failure (message);
  msg (message);
  msg ('Abending.');
  finis (1);
end proc;

#include "vc-admin.setl"
```

A.19 vc-httpd.setl

Service provided:
  httpd
Client of service:
  do (vc-do.setl, Section A.11)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)
  webutil.setl (Section A.44)

Source code:

```
const yhwh = 'vc-httpd.setl';

-- This program thinks it is a little Web server.  It "instantiates"
-- the local file vc-template.html by substituting parameters (some
-- of which typically come from imagemap mouse clicks via a URL) for
-- keywords, and presents the result as HTTP-wrapped HTML.  The latter
-- in turn contains a reference to the server-push JPEG image producer
-- (the push service) and a self-reference that browsers usually
-- decorate with pixel locations when the imagemap is clicked.
--
-- It also sends camera control commands to the do service based on
-- those click parameters and the current state.

const want_nonsense = false;  -- true if you have no Canon hardware

const width = 320;
const height = 240;
--- These constants should go into a file, say vc-limits.setl, which
--- is #included by everything that uses them:
const panlo = -90;
const panhi = 90;
const tiltlo = −30;
const tilthi = 25;
const min_zoom = 1;
const max_zoom = 10;

const pi = 2 ∗ asin 1;
const fmin = −1;
const fmax = 1;
const gmin = −1;
const gmax = 1;
const nsamp = 12;  -- how many segments in the “circle”

-- This flag determines how the camera is controlled during
-- client inactivity:
const method = ‘move’;  -- ‘move’ or ‘speed’

const want_iato = false;  -- want inactivity timeout or not
const tick = if method = ‘move’ then 5000 else 3000 end if;  -- ms
const inactivity_timeout = round (60 ∗ 1000 / tick);  -- 60 sec

const do_fd = fileno if want_nonsense then
  open (getfile ‘nonsense.tcp’, ‘socket’)
else
  obtain_service (‘do’)
end if;

const timer_fd = if want_iato then
  open (str tick, ‘real-ms’)
else
  om
end if;

const sigterm_fd = open (‘SIGTERM’, ‘signal’);  -- catch TERM signals

const httpd_fd = fileno provide_service (‘httpd’);

var clients := {};
```

```
cycling := want_iato;
ticker := 0;
r := 1;  -- starting “radius”

do_ramp (500);
zoom := do_get (‘zoom_factor’);

loop

  [ready] := select ([{sigterm_fd, httpd_fd, timer_fd} +
                                              domain clients]);

  if sigterm_fd in ready then
    msg (yhwh + ‘ (’ + str pid + ‘) caught SIGTERM’);
    quit_gracefully;
  end if;

  for client = clients(pump_fd) | pump_fd in ready loop
    geta (pump_fd, action);
    if action = ‘clicked’ then
      if cycling then
        if method = ‘speed’ then
          do_move_stop;
        end if;
        cycling := false;
      end if;
      reada (pump_fd, zoom_scale, pan, tilt);
      if zoom_scale ≤ 1 then
        do_zoom_by (zoom_scale);
        do_move_by (pan, tilt);
      else
        do_move_by (pan, tilt);
        do_zoom_by (zoom_scale);
      end if;
    end if;
    if action = ‘new’ then
      if want_iato then
        zoom := 1 + random 2.0;
        do_zoom_to (zoom);
        if method = 'speed' then
          do_move_start;
        end if;
        cycling := true;
      end if;
    end if;
    if action = 'initial' then
      if cycling then
        if method = 'speed' then
          do_move_stop;
        end if;
        cycling := false;
      end if;
      reada (pump_fd, old_zoom, zoom, pan, tilt);
      if zoom ≤ old_zoom then  -- like zoom_scale ≤ 1 above
        do_zoom_to (zoom);
        do_move_to (pan, tilt);
      else
        do_move_to (pan, tilt);
        do_zoom_to (zoom);
      end if;
    end if;
    done_client (pump_fd);
  end loop;

  if httpd_fd in ready then  -- new client
    fd := accept (httpd_fd);
    if fd ≠ om then
      name := getname fd;
      msg (name+' accepted');
      [old_pan, old_tilt] := do_get ('position');
      --- confusing use of variable names (zoom, but old_pan etc.):
      zoom := do_get ('zoom_factor');
      pump_fd := pump();
      if pump_fd = −1 then
        -- child (pumping co-process);
        [uri, protocol, mime_headers] := get_request (fd);
```

```
mu := massage_uri uri ? {};
protocol ?:= '';
mu.cmd ?:= 'JPEG';
if mu.click = om then
  mu.top_blurb := 'Try clicking in this image!\n'+
                 '<p>\n\n'+
                 'PWM_EQUIV\n'+
                 '<p>';
  -- Let explicit initial pan, tilt, and zoom specifications
  -- from the URL override the "old" settings that would
  -- otherwise be initially used:
  old_zoom := zoom;  --- is "old" (see "confusing" above)
  zoom := mu.zoom ? old_zoom;
  pan := mu.pan ? old_pan;
  tilt := mu.tilt ? old_tilt;
  mu.pwm := 'Initial pan = '+fixed(pan,0,1)+' deg, '+
                 'tilt = '+fixed(tilt,0,1)+' deg\n<p>'+
           'Initial zoom factor = '+fixed(zoom,0,2)+'\n<p>';
  if mu.zoom ? mu.pan ? mu.tilt = om then
    print ('new');
  else
    print ('initial');
    print (old_zoom, zoom, pan, tilt);
  end if;
else
  --- N.B. For the Canon, I will abuse PWM_EQUIV just to
  --- quote pan and tilt in degrees.
  mu.top_blurb := 'Clicked at:  '+str mu.click+'\n'+
                 '<p>\n\n'+
                 'PWM_EQUIV\n'+
                 '<p>';
  [x,y] := mu.click;
  pan_norm := (x − width/2) / (width/2);
  tilt_norm := (height/2 − y) / (height/2);
  dist_norm := sqrt (pan_norm**2 + tilt_norm**2);
  zoom_scale := 1.618 ** (2 − 4*dist_norm);
  pan := pan_norm * 40 / zoom;
  tilt := tilt_norm * 30 / zoom;
```

```
      -- This string replaces PWM_EQUIV in the HTML template:
      req_pan := old_pan + pan;
      req_tilt := old_tilt + tilt;
      req_zoom := zoom * zoom_scale;
      pan_tilt_clamp_blurb :=
        if req_pan < −90 or req_pan > 90 or
           req_tilt < −30 or req_tilt > 25 then
          '<br>clamped to ['+fixed(req_pan max −90 min 90,0,1)+' '+
                   fixed(req_tilt max −30 min 25,0,1)+']\n'
        else ''
        end if;
      zoom_clamp_blurb :=
        if req_zoom < min_zoom or req_zoom > max_zoom then
          '<br>clamped to '+fixed(req_zoom
                    max min_zoom
                    min max_zoom,0,2)+'\n'
        else ''
        end if;
      mu.pwm := 'Requested pan = '+fixed(req_pan,0,1)+' deg, '+
             'tilt = '+fixed(req_tilt,0,1)+' deg\n'+
            '<br>(delta pan = '+fixed(pan,0,1)+' deg, '+
             'tilt = '+fixed(tilt,0,1)+' deg)\n'+
            pan_tilt_clamp_blurb+'<p>'+
            'Requested zoom factor = '+
              fixed(req_zoom,0,2)+'\n'+
            '<br>('+fixed(zoom,0,2)+' scaled by '+
              fixed(zoom_scale,0,3)+')\n'+
            zoom_clamp_blurb+'<p>';
      print ('clicked');
      print (zoom_scale, pan, tilt);
    end if;
    html := instantiate (getfile 'vc-template.html', mu);
    spew_html (fd, html, protocol);
    stop;
  end if;
  -- parent continues here
  close (fd); -- child deals with this client fd
  client := {};
```

```
    client.pump_fd := pump_fd;
    client.name := name;
    clients(pump_fd) := client;
  end if;
end if;

if timer_fd ≠ om and timer_fd in ready then
  ticker +:= 1;
  geta (timer_fd, dummy);
  if cycling then
    -- Clock the camera through some pattern during client inactivity
    -- (note that this cycling only happens if want_iato = true)
    if ticker mod nsamp = 0 then
      r := 0.2 + random 0.8;
      zoom := 1 + random 4.0;
      do_zoom_to (zoom);
    end if;
    x := (ticker mod nsamp) / nsamp;  -- a real in [0..1)
    y := (pan_cycle (x) − fmin) / (fmax − fmin);  -- also normalized
    z := y ∗ (panhi − panlo) + panlo;  -- scaled to output range
    pan := r ∗ z;
    y := (tilt_cycle (x) − gmin) / (gmax − gmin);  -- also normalized
    z := y ∗ (tilthi − tiltlo) + tiltlo;  -- scaled to output range
    tilt := r ∗ z;
    if method = 'move' then
      s := max_zoom/zoom;
      t := 4/zoom;
      do_move_to (pan + (random s − s/2),
                  tilt + (random t − t/2));
    elseif method = 'speed' then
      do_move_speed (pan/10 + (random 4.0 − 2.0),
                     tilt/10 + (random 1.2 − 0.6));
    end if;
  elseif want_iato and ticker = inactivity_timeout then
    ticker := random nsamp;
    zoom := 1 + random 2.0;
    do_zoom_to (zoom);
    if method = 'speed' then
```

```
        do_move_start;
      end if;
      cycling := true;
    end if;
  end if;

end loop;

proc spew_html (fd, html, protocol);
  if (to_upper protocol)('^HTTP') ≠ om then
    printa (fd, 'HTTP/1.0 200 OK');
    printa (fd, 'Server: WEBeye');
    printa (fd, 'Expires: 0');
    printa (fd, 'Pragma: no-cache');
    printa (fd, 'Content-type: text/html');
    printa (fd, 'Content-length: '+str #html);
    printa (fd);
  end if;
  putc (fd, html);
  flush (fd);
end proc spew_html;

proc instantiate (template, mu);
  gsub (template, 'DATE', fdate(tod));
  gsub (template, 'TOP_BLURB', mu.top_blurb);
  gsub (template, 'PWM_EQUIV', mu.pwm);  -- must go after TOP_BLURB sub.
  gsub (template, 'WEB_HOME', hostaddr);
  -- Look ourselves up, even though port httpd_fd should be the same
  -- as the looked-up httpd port, for consistency with the others (and
  -- as a sort of silly doublecheck):
  gsub (template, 'HTTPD_HOME', public_service ('httpd'));
  gsub (template, 'VIDEO_HOME', public_service ('push'));
  gsub (template, 'CAMERA_TCP', public_service ('camera'));
  gsub (template, '/MAX_RATE', if mu.rate = om then '' else
                                 '/rate='+str mu.rate end);
  return template;
end proc instantiate;
```

```
-- Look up a service name and present it as host:port for the public:
proc public_service (name);
 [serv_name, serv_port] := find_service (name);
 if serv_name = 'localhost' then
  serv_name := hostaddr;
 end if;
 return serv_name+':'+str serv_port;
end proc;

proc pan_cycle (x);
 return sin (2*pi*x);
end;

proc tilt_cycle (x);
 return cos (2*pi*x);
end;

proc new_cmd (name);
 cmd := {};
 cmd.name := name;
 return cmd;
end proc;

proc do_cmd (cmd);
 writea (do_fd, cmd);
 geta (do_fd, response_line);
 return response_line;
end proc;

proc do_ramp (ms);
 cmd := new_cmd ('Ramp');
 cmd.ms := ms;
 do_cmd (cmd);
end proc;

proc do_move_to (pan, tilt);
 do_move ('To', pan, tilt);
end proc;
```

```
proc do_move_by (pan, tilt);
  do_move ('By', pan, tilt);
end proc;

proc do_move (toby, pan, tilt);
  cmd := new_cmd ('Move');
  cmd.subcmd := toby;
  cmd.pan := pan;
  cmd.tilt := tilt;
  do_cmd (cmd);
end proc;

proc do_move_start;
  cmd := new_cmd ('Move');
  cmd.subcmd := 'Start';
  do_cmd (cmd);
end proc;

proc do_move_stop;
  cmd := new_cmd ('Move');
  cmd.subcmd := 'Stop';
  do_cmd (cmd);
end proc;

proc do_move_speed (pan_speed, tilt_speed);
  cmd := new_cmd ('Move');
  cmd.subcmd := 'Speed';
  cmd.pan_speed := pan_speed;
  cmd.tilt_speed := tilt_speed;
  do_cmd (cmd);
end proc;

proc do_zoom_to (factor);
  cmd := new_cmd ('Zoom');
  cmd.subcmd := 'To';
  cmd.zoom_factor := factor;
  do_cmd (cmd);
```

```
end proc;

proc do_zoom_by (scale);
  cmd := new_cmd ('Zoom');
  cmd.subcmd := 'By';
  cmd.zoom_scale := scale;
  do_cmd (cmd);
end proc;

proc do_get (what);
  cmd := new_cmd ('Get');
  cmd.what := what;
  return unstr do_cmd (cmd);
end proc;

proc done_client (pump_fd);
  msg (clients(pump_fd).name + ' done');
  close (pump_fd);
  clients(pump_fd) := om;
end proc done_client;

proc quit_gracefully;
  exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                                          client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
#include "webutil.setl"
```

A.20 vc-image.setl

Service provided:
  image, used by local clients:
    vc-push.setl (Section A.34)
    vc-snap.setl (Section A.41)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-allowed.setl (Section A.2)
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-provide.setl (Section A.32)

Source code:

```
const yhwh = 'vc-image.setl';

-- This server, when it has more than 0 clients connected, keeps an
-- "image pump" maximally busy making images, and sends each new image
-- out to all the ready clients as it arrives.  It also sends this
-- image to clients that become ready before the next one is made.
--
-- This server is intended for "local" use and trusts its clients
-- (probably the children of some higher-level server) to be ready to
-- take a whole image when they send a command line.  (Currently the
-- only command supported is JPEG, and takes no parameters.)  It
-- checks to make sure clients are on the local host, whence the trust.
--
-- Each image is prefaced by a single line of all decimal digits
-- stating the number of bytes in the image that follows.  The client
-- should read that count with reada (or fscanf in C) in case I take
-- the option of appending a newline to the image (which I currently
-- don't, but legally could).
---
--- Now that I am having vc-push.setl, for example, bash away at this
--- thing, I will temporarily (:-) start leaving the image pump open
--- "permanently".
```

```
const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const server_fd = fileno provide_service ('image');

var image_fd := om;  -- image pump fd
var clients := {};

image_errors := 0;
awaiting_image := false;
current_image := '';
image_num := 0;

loop

 [ready] := select ([{sigterm_fd, server_fd, image_fd} +
                                          domain clients]);

 if sigterm_fd in ready then
   msg (yhwh + ' (' + str pid + ') caught SIGTERM');
   quit_gracefully;
 end if;

 for client = clients(client_fd) | client_fd in ready loop
   if (line := getline client_fd) ≠ om and
      #(t := split (line)) = 1 and
      t(1) = 'JPEG' then
     n := #current_image;
     if n > 0 and client.image_num ≠ image_num then
       send_image (client, current_image, image_num);
     else
       client.waiting := true;
     end if;
     clients(client_fd) := client;
   else
     close (client_fd);
     name := client.name;
     image_count := client.image_count;
     -- Restore this message when we get the JPEG "streaming":
```

```
--   time_spent := (clock − client.start_time) / 1000;
--   msg(name+‘ done after ’+fixed(time_spent,0,1)+‘s ’+
--          ‘(’+fixed(image_count / time_spent,0,1)+‘ fps)’);
      clients(client_fd) := om;
    end if;
  end loop;

  if image_fd ≠ om and image_fd in ready then
    reada (image_fd, n);
    if n = om then
      msg (‘image error - n is OM - check /var/log/messages for clues’);
      close (image_fd);
      current_image := ‘’;  -- don’t want to re-use old image after this
      if (image_errors +:= 1) < 10 then
        image_fd := open_image_pump();
        awaiting_image := false;
      else
        msg (str image_errors+‘ image errors in a row - bye!’);
        stop image_errors;
      end if;
    else
      image := getn (image_fd, n);
      if #image ≠ n then
        image_errors +:= 1;
        msg (‘image error - size ’+str #image+‘ /= ’+str n);
      else
        image_errors := 0;
        current_image := image;
        image_num +:= 1;
        for client = clients(client_fd) | client.waiting loop
          send_image (clients(client_fd), current_image, image_num);
        end loop;
      end if;
    end if;
    awaiting_image := false;
  end if;

  if server_fd in ready then
    client_fd := accept (server_fd);
    if client_fd ≠ om then
      name := getname client_fd;
      if allowed (client_fd) then
         -- Restore this message when we get the JPEG “streaming”:
--     msg (name+‘ accepted’);
        clients(client_fd) := new_client (client_fd, name);
      else
        msg (‘untrusted client ’+name+‘ refused’);
        close (client_fd);
      end if;
    end if;
  end if;

  if image_fd = om and #clients > 0 then
    image_fd := open_image_pump();
    awaiting_image := false;
    -- Restore this code when we get the JPEG “streaming”:
--elseif image_fd ≠ om and #clients = 0 then
--  msg (‘closing image pump, image_fd = ’+str image_fd);
--  close (image_fd);
--  image_fd := om;
--  current_image := ‘’;  – it is “old” or soon will be
  end if;

  if image_fd ≠ om and not awaiting_image then
    printa (image_fd,‘JPEG’);
    awaiting_image := true;
  end if;

end loop;

proc open_image_pump();
  image_fd := open (‘image-pump’,‘pump’) ? open (‘busy-pump’,‘pump’);
  if image_fd = om then
    msg (‘cannot open image pump – bye!’);
    quit_gracefully;
  end if;
```

```
  return image_fd;
end proc;

proc new_client (client_fd, name);
  client := {};
  client.client_fd := client_fd;
  client.last_num := 0;
  client.waiting := false;
  client.start_time := clock;
  client.image_count := 0;
  client.name := name;
  return client;
end proc;

proc send_image (rw client, image, image_num);
  client_fd := client.client_fd;
  printa (client_fd, #image);
  putc (client_fd, image);
  flush (client_fd);
  client.image_num := image_num;
  client.image_count +:= 1;
  client.waiting := false;
end proc;

proc quit_gracefully;
  exit_gracefully ([if image_fd = om then om
                   else [str filename image_fd, image_fd]
                   end if]);
end proc;

#include "vc-provide.setl"
#include "vc-getname.setl"
#include "vc-allowed.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.21 vc-init.setl

Called by parent program:
  vc-send.setl (Section A.38)
Calls child program:
  vc-comdev.setl (Section A.7)

Source code:

```
const yhwh = 'vc-init.setl';

-- Initialize the serial device

com_dev := command_line(1) ? filter ('exec setl vc-comdev.setl');

printa (stderr, yhwh, 'begins');

fd := open ('com.settings', 'r');
if fd ≠ om then
  s := getline fd;
  close (fd);
else
  s := '0:0:800004fd:0:3:1c:7f:15:4:0:1:0:11:13:1a:0:12:f:17:16:0:0:73';
end if;
system ('stty ' + s + ' < ' + com_dev);
system ('stty -a < ' + com_dev + ' >&2');

printa (stderr, yhwh, 'ends');
```

A.22 vc-input.setl

Called by parent program:
  vc-recv.setl (Section A.36)
Calls child program:
  vc-comdev.setl (Section A.7)
Textually #includes:
  vc-msg.setl (Section A.30)

Source code:

```
const yhwh = 'vc-input.setl';

-- The existence of this program is predicated on the idea that it is
-- bad to ignore the input serial line for too long, and that when the
-- parent is ready to read the bytes we have so alertly collected, its
-- system-level read is eager to swallow at least as many bytes as
-- have accumulated, up to some absurdly generous limit.  When a Unix
-- pipe fd goes ready to accept output, for example, the typical kernel
-- is prepared to accept 8192 bytes without blocking.
--
-- As for not depending on the kernel to buffer up lots of input bytes
-- if you get unthinkably behind, that probably is excessive paranoia
-- in retrospect.  So you could probably do without this program, and
-- just have vc-recv.setl read directly from the device.  Meanwhile,
-- it's nice to know that it can probably buffer up to 8 seconds' worth
-- of bytes at 1000/sec and never spend so long writing that out that
-- it ignores the input for very long.

-- This program is normally invoked from vc-input, which is
-- simply a setuid'd wrapper compiled from a C program containing
--
-- main() {
--  execl("$(SETL)", "setl", "vc-input.setl", 0);
-- }
--
-- where $(SETL) has been substituted with the absolute pathname of
-- the 'setl' program (the SETL driver) by the Makefile.
```

```
com_dev := command_line(1) ? filter ('exec setl vc-comdev.setl');
com_fd := fileno open (com_dev, 'r');
s := '';
ready_for_output := false;
loop
  input_pool := {com_fd};
  output_pool := if ready_for_output then {} else {stdout} end if;
  [in_ready, out_ready] := select ([input_pool, output_pool]);
  if #in_ready > 0 then
    c := getc (com_fd);
    if c ≠ om then
      s +:= c;
    else
      -- On EOF (which "clear" can cause), just close and reopen
      close (com_fd);
      select (om, 300);  -- wait 300 ms before reopening
      com_fd := fileno open (com_dev, 'r');
    end if;
  end if;
  if #out_ready > 0 then
    ready_for_output := true;
  end if;
  if #s > 0 and ready_for_output then
    putchar (s);
    flush (stdout);
    s := '';
    ready_for_output := false;
  end if;
end loop;

#include "vc-msg.setl"
```

A.23 vc-javent.setl

Textually #included by:
  vc-evjump.setl (Section A.13)
  vc-evzoom.setl (Section A.14)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)

Source code:

```
-- This file is meant to be #included by others, after they define:
--
-- yhwh  -  the name of the program that #includes this code
-- event_type  -  the name of the service being provided
-- of_interest  -  the command tokens we look for
-- n_parms  -  how many parameters after the command tokens to expect

const n = #of_interest;

const notice_fd = fileno obtain_service ('notice');
const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const server_fd = fileno provide_service (event_type);

var clients := {};
var current_event := om;

open ('SIGPIPE', 'ignore');  -- as when we write to closed observers

loop

 [ready] := select ([{sigterm_fd, server_fd, notice_fd} +
                                      domain clients]);

 if sigterm_fd in ready then
   msg (yhwh + ' (' + str pid + ') caught SIGTERM');
```

```
  quit_gracefully;
end if;

for client = clients(pump_fd) | pump_fd in ready loop
  ack := getline pump_fd;
  if eof then
    done_client (pump_fd);
  else
    if client.last_event ≠ current_event then
      tell_client (pump_fd, current_event);
    else
      clients(pump_fd).hungry := true;
    end if;
  end if;
end loop;

if server_fd in ready then
  fd := accept (server_fd);
  if fd ≠ om then
    name := getname fd;
    msg (name+‘ accepted’);
    pump_fd := pump();
    if pump_fd = −1 then
      -- child
      while (event := getline stdin) ≠ om loop
        printa (fd, event);
        if (ack := getline fd) = om then
          stop;
        end if;
        print;
      end loop;
      stop;
    end if;
    -- parent continues here
    close (fd);
    client := {};
    client.name := name;
    client.hungry := true;
```

```
      clients(pump_fd) := client;
      -- send initial event if any
      if current_event ≠ om then
        tell_client (pump_fd, current_event);
      end if;
    end if;
  end if;

  if notice_fd in ready then
    reada (notice_fd, raw_event);
    if is_string raw_event then
      t := split (raw_event);
      if t(1 .. n) = of_interest and #t−n = n_parms then
        tell_observers ((+/[‘ ’+x : x in t(n+1 .. )])(2 .. ));
      end if;
    end if;
  end if;

end loop;

proc tell_observers (event);
  for client = clients(pump_fd) loop
    if client.hungry then
      tell_client (pump_fd, event);
    end if;
  end loop;
  current_event := event;
end proc tell_observers;

proc tell_client (pump_fd, event);
  printa (pump_fd, event);
  flush (pump_fd);
  if last_error ≠ no_error then
    done_client (pump_fd);
  else
    client := clients(pump_fd);
    client.hungry := false;
    client.last_event := event;
    clients(pump_fd) := client;
  end if;
end proc tell_client;

proc done_client (pump_fd);
  msg (clients(pump_fd).name + ' done');
  close (pump_fd);
  clients(pump_fd) := om;
end proc done_client;

proc quit_gracefully;
  exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                                        client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.24 vc-jmaster.cgi

Source code:

```
#! SETL_BIN/setl -k

-- This script should be placed in the Web server's cgi-bin directory or
-- equivalent, after SETL_BIN and VC_DIR are defined by configuration.

const vc_dir  = VC_DIR;  -- WEBeye home directory name

const vc_link = vc_dir+'/vc-link.html';
const fresh   = vc_dir+'/vc-fresh.html';
const broken  = vc_dir+'/vc-broken.html';
const java    = vc_dir+'/vc-java.html';

const vc_prefix  = 'CGI|';  -- pseudo-document convention

magic := false;

-- Try up to 10 times to read through link.  The link file should
-- always exist after the first time the Box is started, except
-- very briefly during major life-cycle transitions when the Box is
-- shut down or restarted:
loop for i in {1 .. 10} while (content := getfile vc_link) = om do
  select (om, 50);  -- wait 50 ms and try again
end loop;

if content = om then
  -- WEBeye Box probably never started
  put_document (getfile fresh);

elseif match (content, vc_prefix) = vc_prefix then
  -- Pseudo-document
  lookup := break (content, '|');  -- location of lookup service
  lookup_fd := open (lookup, 'socket');
  if lookup_fd = om then
    -- Lookup server did not accept connection
    put_diagnostic_about ('lookup');
  else
    html := getfile java;
    for service_name in ['giver', 'mouse', 'evjump', 'evzoom'] loop
      writea (lookup_fd, service_name);
      reada (lookup_fd, [service_host, service_port]);
      gsub (html, to_upper service_name + '_PORT', str service_port);
    end loop;
    close (lookup_fd);
    put_document (html);
  end if;

else
  -- Link points to a (static) real document
  put_document (content);
end if;

proc put_mime_headers;
  print ('Content-type: text/html');
  print;
end proc;

proc put_document (content);
  put_mime_headers;
  putchar (content);
end proc;

proc put_diagnostic_about (service_name);
  var content := getfile broken;
  gsub (content, 'SERVICE', service_name);
  put_document (content);
end proc;
```

A.25 vc-jumper.setl

Service provided:
  jumper
Client of service:
  do (vc-do.setl, Section A.11)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-simpler.setl (Section A.40)

Source code:

```
const yhwh = 'vc-jumper.setl';

-- Simplified "Jump To" command interface.

-- The client just sends a pair of numbers (pan and tilt) on each line.

-- The name of the service we provide:
#define service_name 'jumper'

-- The number of parameters on the command:
#define n_parms  2

-- The full details of the command we will send to the do service:
#define build_cmd  \
  cmd := {};  \
  cmd.name := 'Jump';  \
  cmd.subcmd := 'To';  \
  cmd.pan := val t(1);  \
  cmd.tilt := val t(2);

#include "vc-simpler.setl"
```

A.26 vc-master.cgi

Source code:

```
#! SETL_BIN/setl -k

-- This script should be placed in the Web server's cgi-bin directory or
-- equivalent, after SETL_BIN and VC_DIR are defined by configuration.

const vc_dir  = VC_DIR;  -- WEBeye home directory name

const vc_link = vc_dir+'/vc-link.html';
const fresh   = vc_dir+'/vc-fresh.html';
const broken  = vc_dir+'/vc-broken.html';

const vc_prefix  = 'CGI|';  -- pseudo-document convention

magic := false;

-- Try up to 10 times to read through link.  The link file should
-- always exist after the first time the Box is started, except
-- very briefly during major life-cycle transitions when the Box is
-- shut down or restarted:
loop for i in {1 .. 10} while (content := getfile vc_link) = om do
  select (om, 50);  -- wait 50 ms and try again
end loop;

if content = om then
  -- WEBeye Box probably never started
  put_document (getfile fresh);

elseif match (content, vc_prefix) = vc_prefix then
  -- Pseudo-document
  lookup := break (content, '|');  -- location of lookup service
  lookup_fd := open (lookup, 'socket');
  if lookup_fd = om then
    -- Lookup server did not accept connection
    put_diagnostic_about ('lookup');
```

```
 else
  writea (lookup_fd, ‘httpd’);
  reada (lookup_fd, [httpd_host, httpd_port]);
  close (lookup_fd);
  httpd := httpd_host + ‘:’ + str httpd_port;
  httpd_fd := open (httpd, ‘socket’);
  if httpd_fd = om then
   -- WEBeye httpd server did not accept connection
   put_diagnostic_about (‘httpd’);
  else
   -- Construct HTTP request and send to WEBeye httpd
   query := getenv ‘PATH_INFO’ + ‘?’ + getenv ‘QUERY_STRING’;
   printa (httpd_fd, ‘GET’, query, ‘HTTP/1.0’);
   printa (httpd_fd);
   if ‘HTTP/1.0 200 OK’ in (getline httpd_fd ? ‘’) then
    -- Success - serve document returned by WEBeye httpd
    putchar (getfile httpd_fd);
   else
    -- WEBeye httpd up but not responding properly
    put_diagnostic_about (‘httpd’);
   end if;
   close (httpd_fd);
  end if;
 end if;

else
 -- Link points to a (static) real document
 put_document (content);
end if;

proc put_mime_headers;
 print (‘Content-type: text/html’);
 print;
end proc;

proc put_document (content);
 put_mime_headers;
 putchar (content);
```

```
end proc;

proc put_diagnostic_about (service_name);
  var content := getfile broken;
  gsub (content, 'SERVICE', service_name);
  put_document (content);
end proc;
```

A.27 vc-model.setl

Called by parent program:
  vc-do.setl (Section A.11)
Calls child program:
  vc-seq.setl (Section A.39)
Textually #includes:
  vc-decode.setl (Section A.10)
  vc-exit.setl (Section A.15)
  vc-msg.setl (Section A.30)

Source code:

```
const yhwh = 'vc-model.setl';

-- This is the main pump used by the do server, and maintains a
-- high-level model of the pan/tilt/zoom system.
--
-- It takes requests already checked and encapsulated as SETL maps,
-- performs them, and replies with similarly encapsulated responses.
--
-- It processes requests sequentially, but internally uses a "sequencer"
-- program, vc-seq.setl, to take advantage of the possibility
-- of overlapped command-and-response (full duplex) communications with
-- the Canon VC-C3 to implement speed-ramped motion trajectories in
-- which pan/tilting and zooming are performed simultaneously.

-- Command sequencer:
const seq_fd = fileno open ('exec setl vc-seq.setl', 'pump');

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals

var cur_mode := 'RC';  -- by assumption, but init_model changes it

var cur_zoom_factor := 1;
var cur_zoom_speed := 0;
var currently_zooming := false;

var cur_pan := 0, cur_tilt := 0;
var cur_pan_speed := 0, cur_tilt_speed := 0;
var currently_moving := false;

var ms_per_tick := 100;
const min_move_speed = 1;
const max_move_speed = 70;
const max_speed = max_move_speed * ms_per_tick / 1000;  -- deg/tick
const max_zoom_speed = 8;  -- the “min” is the -ve, and zooms out

const max_pan_speed = 76;  -- the “min” is the -ve, and pans left
const max_tilt_speed = 70;  -- the “min” is the -ve, and tilts down

const min_zoom = 1;
const max_zoom = 10;

var cur_ramp := 500;  -- ms
var ramp_ticks := ms_to_ticks cur_ramp;
var accel := max_speed / (1 max ramp_ticks);  -- speed change per tick

tie (stdin, stdout);

init_model;

loop doing
  [ready] := select ([{sigterm_fd, stdin}]);
  if sigterm_fd in ready then
    msg (yhwh + ‘ (’ + str pid + ‘) caught SIGTERM’);
    quit_gracefully;
  end if;
  read (cmd);
while not eof do
  write (perform (cmd));  -- perform cmd and give client the response
end loop;

proc init_model;
  do_init;
  do_mode_host;
  do_zoom_stop;
  do_move_stop;
  do_zoom_speed (5);  -- a medium-high default speed for zooms
end proc;

proc perform (cmd);
  r := do_response ([ ]);  -- default
  case cmd.name of
  (‘Zoom’):
    case cmd.subcmd of
    (‘Start’):  r := do_zoom_start();
    (‘Stop’):   r := do_zoom_stop();
    (‘Speed’):  r := do_zoom_speed (cmd.zoom_speed);
    (‘To’):     r := do_zoom_to (cmd.zoom_factor, cmd.speed);
    (‘By’):     r := do_zoom_by (cmd.zoom_scale,  cmd.speed);
    else msg (‘Unrecognized Zoom subcmd ’+str cmd.subcmd);
    end case;
  (‘Move’):
    case cmd.subcmd of
    (‘Start’):  r := do_move_start();
    (‘Stop’):   r := do_move_stop();
    (‘Speed’):  r := do_move_speed (cmd.pan_speed, cmd.tilt_speed);
    (‘To’):     r := do_move_to (cmd.pan, cmd.tilt, cmd.ms, cmd.speed);
    (‘By’):     r := do_move_by (cmd.pan, cmd.tilt, cmd.ms, cmd.speed);
    else msg (‘Unrecognized Move subcmd ’+str cmd.subcmd);
    end case;
  (‘Jump’):
    case cmd.subcmd of
    (‘To’):  r := do_jump_to (cmd.pan, cmd.tilt);
    --- still to implement:  ’By’
    else msg (‘Unrecognized Jump subcmd ’+str cmd.subcmd);
    end case;
  (‘Ramp’):
    r := do_ramp (cmd.ms);
  (‘Mode’):
    case cmd(‘mode’) of  -- i.e., ‘cmd.mode’, but mode is a keyword
    (‘Host’):  r := do_mode_host();
    (‘RC’):    r := do_mode_rc();
    else msg (‘Unrecognized mode ’+str cmd(‘mode’));
    end case;
  ('Clear'):
    r := do_clear();
  ('Reload'):
    r := do_reload();
  ('Setup'):
    r := do_setup();
  ('Reset'):
    r := do_reset();
  ('Check'):
    r := do_check();
  ('Hex'):
    r := do_hex (cmd.cmd);
  ('Get'):
    r := do_get (cmd.what);
  else
    msg ('Unrecognized cmd name '+str cmd.name);
  end case;
  return r;
end proc perform;

proc do_zoom_start();
  [dev_dir, dev_speed] := downcvt_zoom_speed (cur_zoom_speed);
  if dev_dir = '\xff' then  -- zero zoom speed ⇒ stop zooming
    return do_zoom_stop();
  end if;
  if not currently_zooming then
    do_cmd (unhex '011201'          -- device Start TELE or WIDE
              + dev_dir);
    currently_zooming := true;
  end if;
  return do_response (['Zoom Start']);
end proc do_zoom_start;

proc do_zoom_stop();
  if currently_zooming then
    do_cmd (unhex '011203');          -- device Zoom Stop
    currently_zooming := false;
  end if;
  return do_response (['Zoom Stop']);
end proc do_zoom_stop;

proc do_zoom_speed (zoom_speed);
  zoom_speed max:= −max_zoom_speed;
  zoom_speed min:= max_zoom_speed;
  [dev_dir, dev_speed] := downcvt_zoom_speed (zoom_speed);
  if dev_dir = '\xff' then  -- zero zoom speed ⇒ stop zooming
    if currently_zooming then
      do_zoom_stop();
    end if;
  else
    if currently_zooming and
       dev_dir ≠ (downcvt_zoom_speed (cur_zoom_speed))(1) then
      -- a "sign change" in the zoom speed
      do_cmd (unhex '011203');         -- device Zoom Stop
      do_cmd (unhex '01120402'         -- device Zoom Speed
                  + char dev_speed);
      do_cmd (unhex '011201'           -- device Zoom Start
                  + dev_dir);
    else
      do_cmd (unhex '01120402'         -- device Zoom Speed
                  + char dev_speed);
    end if;
  end if;
  cur_zoom_speed := zoom_speed;
  return do_response (['Zoom Speed ' + fixed (zoom_speed, 0, 1)]);
end proc do_zoom_speed;

proc do_zoom_to (zoom_factor, at_speed);
  if currently_zooming then
    do_zoom_stop();
  end if;
  if at_speed ≠ om then
    do_zoom_speed (at_speed);
  end if;
  zoom_factor max:= min_zoom;
```

```
 zoom_factor min:= max_zoom;
 dev_factor := downcvt_zoom_factor (zoom_factor);
 -- 3 to 12 seconds, depending:
 time_limit := 1000 * abs (zoom_factor − cur_zoom_factor) /
                        (1 max abs cur_zoom_speed) + 3000;
 do_cmd (unhex '01120202'            -- device Zoom To
           + to_two_bytes dev_factor, time_limit);
 cur_zoom_factor := zoom_factor;
 return do_response (['Zoom ' + fixed (zoom_factor, 0, 3) +
        if at_speed ≠ om then ' At ' + whole (cur_zoom_speed, 0)
        else ''
        end if]);
end proc do_zoom_to;

proc do_zoom_by (zoom_scale, at_speed);
 return do_zoom_to (zoom_scale * cur_zoom_factor, at_speed);
end proc do_zoom_by;

proc do_move_start();
 [dev_pan_dir, dev_pan_speed] := downcvt_pan_speed (cur_pan_speed);
 [dev_tilt_dir, dev_tilt_speed] := downcvt_tilt_speed (cur_tilt_speed);
 if dev_pan_dir = '\x00' and  -- zero speeds ⇒ stop panning
    dev_tilt_dir = '\x00' then
   return do_move_stop();
 end if;
 if not currently_moving then
   do_cmd (unhex '051201'          -- device Pan/Tilt Start
             + dev_pan_dir
             + dev_tilt_dir);
   currently_moving := true;
 end if;
 return do_response (['Move Start']);
end proc do_move_start;

proc do_move_stop();
 if currently_moving then
   do_cmd (unhex '051202');         -- device Pan/Tilt Stop
   currently_moving := false;
```

```
  end if;
  return do_response ([‘Move Stop’]);
end proc do_move_stop;

proc do_move_speed (pan_speed, tilt_speed);
  pan_speed max:= −max_pan_speed;
  pan_speed min:= max_pan_speed;
  tilt_speed max:= −max_tilt_speed;
  tilt_speed min:= max_tilt_speed;
  [dev_pan_dir, dev_pan_speed] := downcvt_pan_speed (pan_speed);
  [dev_tilt_dir, dev_tilt_speed] := downcvt_tilt_speed (tilt_speed);
  if dev_pan_dir = ‘\x00’ and  -- zero speeds ⇒ stop panning
     dev_tilt_dir = ‘\x00’ then
    if currently_moving then
      do_move_stop();
    end if;
  else
    [cur_dev_pan_dir, −] := downcvt_pan_speed (cur_pan_speed);
    [cur_dev_tilt_dir, −] := downcvt_tilt_speed (cur_tilt_speed);
    if currently_moving and
       (dev_pan_dir ≠ cur_dev_pan_dir or
        dev_tilt_dir ≠ cur_dev_tilt_dir) then
      -- a “sign change” in the pan and/or tilt speed
      do_cmd (unhex ‘051202’);              -- device Pan/Tilt Stop
      do_cmd (unhex ‘05120302’               -- device Pan/Tilt Speed
                    + char dev_pan_speed
                    + char dev_tilt_speed);
      do_cmd (unhex ‘051201’                 -- device Pan/Tilt Start
                    + dev_pan_dir
                    + dev_tilt_dir);
    else
      -- not moving, or no sign change; a simple speed change will do
      do_cmd (unhex ‘05120302’               -- device Pan/Tilt Speed
                    + char dev_pan_speed
                    + char dev_tilt_speed);
    end if;
  end if;
  cur_pan_speed := pan_speed;
  cur_tilt_speed := tilt_speed;
  return do_response (['Move Speed ' + fixed (pan_speed, 0, 1) +
                       ' ' + fixed (tilt_speed, 0, 1)]);
end proc do_move_speed;

proc do_move_to (pan, tilt, ms, at_speed);
  assert ms = om or at_speed = om;  -- or both
  if currently_moving then
    do_move_stop();
  end if;
  -- It's no good planning trajectories to places we can't go, so
  -- clamp the request in case the caller hasn't bothered to deal with
  -- that yet, using the published limits:
  pan max:= −90;
  pan min:= +90;
  tilt max:= −30;
  tilt min:= +25;
  dpan := pan − cur_pan;
  dtilt := tilt − cur_tilt;
  deg := sqrt (dpan**2 + dtilt**2);
  ymid := deg / 2;
  -- The time axis is called x here, often indexed by integer k
  if ymid > 0 then
    dircos := dpan / deg;
    dirsin := dtilt / deg;
    ramp_ticks := ms_to_ticks cur_ramp;
    accel := max_speed / (1 max ramp_ticks);  -- speed change per tick
    n := 1 max ceil ramp_ticks;  -- a point at the ramp end or beyond
    -- Obtain xmid
    if at_speed ≠ om then
      at_speed max:= min_move_speed;
      at_speed min:= max_move_speed;
      speed := at_speed * ms_per_tick/1000;  -- deg/sec → deg/tick
      assert speed > 0;
      assert exists k in [n−1,n−2 .. 0] | k*accel < speed;
      speed min:= (k+1)*accel;
      xmid := k + (ymid−max_traj(k)) / speed;
      xmid := (ceil (2 * xmid)) / 2;
```

```
else
  trajtime := ms ? 0;
  trajtime max:= 0;
  trajtime min:= 200000;  -- 200 deg at 1 deg/sec = 200 sec
  xmid := (ceil ms_to_ticks trajtime) / 2;
  if ymid > max_traj(xmid) then
    -- We cannot get there fast enough; revise xmid eastward
    -- accordingly by setting xmid to where the max-speed
    -- trajectory (see proc max_traj) reaches ymid and then
    -- rounding xmid up to the nearest half-tick
    assert exists k in [n−1,n−2 .. 0] | max_traj(k) < ymid;
    speed := ((k+1)∗accel) min max_speed;
    xmid := k + (ymid−max_traj(k)) / speed;
    xmid := (ceil (2 ∗ xmid)) / 2;
    -- Now we can reach ymid in this upwardly revised time xmid
  end if;
end if;
-- xmid was or is okay now
m := (n min ceil xmid) − 1;
assert m ≥ 0;  -- else degenerate case slipped through
-- Find the point (x = k) of departure from the max traj
assert exists k in [m,m−1 .. 0] | traj(xmid,k,k∗accel) < ymid;
-- Mid-trajectory speed is slope going from there up to xmid, ymid
speed := (ymid−max_traj(k)) / (xmid−k);
-- Trajectory function is now max_traj (x) for x ≤ k, and
-- traj (x, k, speed) for x ≥ k (when x = k, take your pick)
ramp_cmds := [move_speed_cmd (accel∗x, dircos, dirsin) :
                                 x in [1 .. k]];
x_end := round (2 ∗ xmid);  -- 2 ∗ xmid already integer in theory
cmds := [[x−1,     ramp_cmds (x)] : x in [1 .. k]] +
        [[k,       move_speed_cmd (speed, dircos, dirsin)]] +
        [[x_end−x, ramp_cmds (x)] : x in [k,k−1 .. 1]] +
        [[x_end,   move_stop_cmd()]];
cmds(2 .. 1) := [[0, move_start_cmd (speed, dircos, dirsin)]];
-- cmds = [[ticknum, cmd], ...]
time_limit := 2000;  -- allow 2 sec after final Stop
responses := do_cmds (cmds, ms_per_tick, time_limit);
if #responses ≠ #cmds then
```

```
   msg (str #cmds+‘ commands sent but ’+str #responses+
                                        ‘ responses received’);
  else
   for response = responses(i) loop
    [−,cmd] := cmds(i);
    if not (response satisfies cmd) then
     report (response, cmd);
    end if;
   end loop;
  end if;
 end if ymid;
 -- Have the camera “settle” to its final position at a speed
 -- suitable for the current zoom factor:
 zorp (pan, tilt, max_zoom / cur_zoom_factor);
 return do_response ([‘Move To ’ + fixed (pan, 0, 3) +
                        ‘ ’ + fixed (tilt, 0, 3) +
       if at_speed ≠ om then ‘ At ’ + whole (at_speed, 0)
       elseif ms ≠ om then ‘ In ’ + whole (ms, 0)
       else ‘’
       end if]);
end proc do_move_to;

proc do_move_by (dpan, dtilt, ms, at_speed);
 return do_move_to (cur_pan + dpan, cur_tilt + dtilt, ms, at_speed);
end proc do_move_by;

proc move_speed_cmd (speed, dircos, dirsin);
 deg_per_sec := speed ∗ 1000/ms_per_tick;
 [−, dev_pan_speed] := downcvt_pan_speed (deg_per_sec ∗ dircos);
 [−, dev_tilt_speed] := downcvt_tilt_speed (deg_per_sec ∗ dirsin);
 return unhex ‘05120302’            -- device Pan/Tilt Speed
          + char dev_pan_speed
          + char dev_tilt_speed;
end proc;

proc move_start_cmd (speed, dircos, dirsin);
 deg_per_sec := speed ∗ 1000/ms_per_tick;
 [dev_pan_dir, −] := downcvt_pan_speed (deg_per_sec ∗ dircos);
 [dev_tilt_dir, −] := downcvt_tilt_speed (deg_per_sec ∗ dirsin);
 return unhex '051201'              -- device Pan/Tilt Start
          + dev_pan_dir
          + dev_tilt_dir;
end proc;

proc move_stop_cmd();
 return unhex '051202';             -- device Pan/Tilt Stop
end proc;

proc do_jump_to (pan, tilt);
 zorp (pan, tilt, max_move_speed);
 return do_response (['Move To ' + fixed (pan, 0, 3) +
                     ' ' + fixed (tilt, 0, 3)]);
end proc do_jump_to;

-- Utility for do_move_to, do_jump_to
 do_move_status(); -- reload cur_pan, cur_tilt from hardware
 dpan := pan − cur_pan;
 dtilt := tilt − cur_tilt;
 deg := sqrt (dpan∗∗2 + dtilt∗∗2);
 if deg > 0 then
   dircos := dpan / deg;
   dirsin := dtilt / deg;
   -- Base the time limit on the reasonable assumption of greater error
   -- over longer trajectories:
   do_move_speed (settle_speed ∗ dircos, settle_speed ∗ dirsin);
   settle_time := 1000 + 200 ∗ deg / settle_speed;
   dev_pan := downcvt_pan (pan);
   dev_tilt := downcvt_tilt (tilt);
   do_cmd (unhex '05120502'           -- device Pan/Tilt To
             + to_two_bytes dev_pan
             + to_two_bytes dev_tilt, settle_time);
   cur_pan := pan;
   cur_tilt := tilt;
 end if;
end proc zorp;
```

```
proc do_ramp (ms);
  ms max:= 0;
  ms min:= 15000;  -- even a 15-second ramp is pretty incredibly slow
  cur_ramp := ms;
  return do_response ([‘Ramp ’ + str ms]);
end proc do_ramp;

proc do_mode_host();
  do_clear;  -- toggle RTS
  do_cmd (unhex ‘08170100’);    -- device Mode Select PC
  cur_mode := ‘Host’;
  -- N.B. do_setup auto-detects home and re-reads pan/tilt/zoom info:
  return do_response ([‘Mode Host’] + do_setup().notices);
end proc do_mode_host;

proc do_mode_rc();
  do_clear;  -- toggle RTS
  do_cmd (unhex ‘08170101’);    -- device Mode Select Remote Controller
  cur_mode := ‘RC’;
  return do_response ([‘Mode RC’]);
end proc do_mode_rc;

proc do_init();
  -- Condition the serial line
  do_cmd (‘i’);  -- “initialize”
  return do_response ([ ]);
end proc do_init;

proc do_clear();
  -- Lower the RTS line for 100 ms, then raise it
  do_cmd (‘c’);  -- “clear”
  return do_response ([ ]);
end proc do_clear;

proc do_reload();
  --
  -- Note that the model’s impression of whether the platform is
  -- currently pantilting or zooming is not updated by this reload,
  -- since I don't know how to read those bits from the hardware.
  --
  -- Moreover, since directions are bound up with the zoom and
  -- pan/tilt Start subcommands, I can't read those either, so my
  -- update of the zoom and pan and tilt speeds just assumes the
  -- sign is whatever I already think it is.
  --
  -- Similarly I must shrug and not update the current control mode
  -- (Host or RC).
  --
  return do_response (do_zoom_status().notices +
                      do_move_status().notices);
end proc do_reload;

proc do_zoom_status();
  cmd := unhex '010402';             -- device Zoom Status
  response := do_cmd (cmd);
  if is_string response and #response = 6 and response(2) = '\x84' then
   dev_zoom_speed  := abs response(4);
   dev_zoom_factor := from_two_bytes response(5 .. 6);
   [dev_zoom_dir, -] := downcvt_zoom_speed (cur_zoom_speed);
   cur_zoom_speed  := upcvt_zoom_speed (dev_zoom_dir, dev_zoom_speed);
   cur_zoom_factor := upcvt_zoom_factor (dev_zoom_factor);
   return do_response (['Zoom Speed ' + fixed (cur_zoom_speed, 0, 1),
                        'Zoom '       + fixed (cur_zoom_factor, 0, 3)]);
  end if;
  msg ('Unexpected response ' + str decode response +
     ' to Zoom Status command ' + str decode cmd);
  return do_reset_if_negative_response (response);
end proc do_zoom_status;

proc do_move_status();
  cmd := unhex '050402';             -- device Pan/Tilt Status
  response := do_cmd (cmd);
  if is_string response and #response = 9 and response(2) = '\x84' then
   dev_pan_speed  := abs response(4);
   dev_tilt_speed := abs response(7);
   dev_pan        := from_two_bytes response(5 .. 6);
   dev_tilt       := from_two_bytes response(8 .. 9);
   [dev_pan_dir, −] := downcvt_pan_speed (cur_pan_speed);
   cur_pan_speed  := upcvt_pan_speed (dev_pan_dir, dev_pan_speed);
   [dev_tilt_dir, −] := downcvt_tilt_speed (cur_tilt_speed);
   cur_tilt_speed := upcvt_tilt_speed (dev_tilt_dir, dev_tilt_speed);
   cur_pan        := upcvt_pan (dev_pan);
   cur_tilt       := upcvt_tilt (dev_tilt);
   return do_response ([‘Move Speed ’ + fixed (cur_pan_speed, 0, 1) +
                               ‘ ’ + fixed (cur_tilt_speed, 0, 1),
                        ‘Move To ’ + fixed (cur_pan, 0, 3) +
                             ‘ ’ + fixed (cur_tilt, 0, 3)]);
  end if;
  msg (‘Unexpected response ’ + str decode response +
     ‘ to Pan/Tilt Status command ’ + str decode cmd);
  return do_reset_if_negative_response (response);
end proc do_move_status;

proc do_reset_if_negative_response (frame);
  -- This is a hack, obviously.  It turns out that when the Canon
  -- starts responding to Status requests with “negative response”,
  -- it will continue to do so.  Whether this is dB’s fault or a glitch
  -- in the Canon firmware remains unknown.  Fortunately, it happens
  -- rarely, much less than once per day in the early months of testing.
  -- Whatever the etiology, a remedy that works is to pull out the
  -- hammer and do a Reset whenever this condition is detected.
  if #frame ≥ 2 and
    (abs frame(1) bit_and 16#80) = 0 and
    (abs frame(2) bit_and 16#60) = 16#40 then
   return do_reset();
  else
   return do_response ([ ]);
  end if;
end proc;

proc do_setup();
  cmd := unhex ‘0510’;               -- device Pan/Tilt Setup
  response := do_cmd (cmd, 5000);  -- 5 sec time limit
  if is_string response and #response = 6 and response(2) = ‘\x90’ then
```

```
    pass;  -- response(3 .. 6) has the absolute position, but
           -- do_reload will pick that up in a moment anyway
  else
    msg (‘Unexpected response ’ + str decode response +
         ‘ to Pan/Tilt Setup command ’ + str decode cmd);
  end if;
  return do_reload();  -- refresh model state from hardware state
end proc do_setup;

proc do_reset();
  do_init;
  do_cmd (‘a’, 6000);  -- provoke VC-C3 auto-init, 6 sec time limit
  return do_mode_host();
end proc do_reset;

proc do_check();
  zoom_factor := cur_zoom_factor;
  [pan, tilt] := [cur_pan, cur_tilt];
  do_reload();
  --- move these fuzz constants out to a more respectable place:
  if abs (cur_pan − pan) > 0.25 or
     abs (cur_tilt − tilt) > 0.25 or
     --- It’s a fiction that this is a zoom “factor”.
     --- It almost certainly is just the position of the servo
     --- controlling the zoom ring:
     abs (cur_zoom_factor − zoom_factor) > 0.05 then
    return do_reset();
  end if;
  return do_response ([ ]);
end proc do_check;

proc do_hex (cmd);  -- unadvertised, for low-level experimentation
  response := do_cmd (unhex cmd, 20000);  -- 20-second time limit (wow)
  return do_response ([‘Frame ’ + str decode response]);
end proc do_hex;

proc do_get (what);
  r := do_response ([ ]);
  r.value := case what of
  ('mode'):        cur_mode,
  ('zoom_factor'): cur_zoom_factor,
  ('zooming'):     currently_zooming,
  ('zoom_speed'):  cur_zoom_speed,
  ('position'):    [cur_pan, cur_tilt],
  ('moving'):      currently_moving,
  ('move_speed'):  [cur_pan_speed, cur_tilt_speed],
  ('ramp'):        cur_ramp
  else msg ('Unrecognized Get argument '+str what)
  end case;
  return r;
end proc do_get;

-- The model replies to the client with a record including 'notices':
proc do_response (notices);
  r := {};
  r.notices := notices;
  return r;
end proc;


proc do_cmd (cmd, time_limit(*));  -- do command and get lo-lev response
  seq := {};
  seq.cmd := cmd;
  -- 3-second response timeout default:
  seq.time_limit := time_limit(1) ? 3000;
  return do_step (seq);
end proc;

proc do_cmds (cmds, tick_ms, time_limit(*));  -- many cmds, responses
  seq := {};
  seq.cmds := cmds;
  seq.tick_ms := tick_ms;
  -- 4.5-second response timeout default:
  seq.time_limit := time_limit(1) ? 4500;
  return do_step (seq);
end proc;
```

```
proc do_step (seq);  -- send a command-sequence packet and get response
  writea (seq_fd, seq);
  reada (seq_fd, response);
  if eof then
    msg (yhwh + ' got EOF from sequencer - quitting');
    quit_gracefully;
  end if;
  return response;
end proc;

op satisfies (response, cmd);
  assert is_string cmd;
  if response = cmd then return true; end if;
  if not is_string response then return false; end if;
  if #response ≥ 2 and #cmd ≥ 2 and
     response(1) + char (abs response(2) bit_and 16#3f#) +
     response(3 .. #response min #cmd) = cmd then
    return true;
  end if;
  return false;
end op;

proc report (response, cmd);
  if not is_string response then
    msg ('Non-string response '+str response+
             ' to command '+str decode cmd);
  else
    msg ('Unexpected response '+str decode response+
             ' to command '+str decode cmd);
  end if;
end proc;


-- General note about speed conversions:  "down" conversions always
-- yield a sign as a single character suitable for plugging into a
-- device-level command, and a magnitude that is a positive integer
-- in the range 0 .. 65535 suitable for passing to two_bytes.
-- Zoom rate conversions:
--
-- The zoom hardware requires a direction and a 3-bit magnitude.
--
-- But since that does not provide for a zero, I will use as
-- the “direction” a value of ‘\xff’ rather than one of the valid
-- values ‘\x00’ (TELE) or ‘\x01’ (WIDE) if I think the zoom speed
-- you supply is about 0.
--
-- I don’t yet know the relationship between the 8 available speeds
-- in each direction and the actual change in zoom factor per second,
-- so for now the mapping is just this:
--
-- I take your zoom_speed spec and map it to the nearest integer n
-- in −8 .. 8.  If n is 0, you get dev_zoom_dir = ‘\xff’ (and
-- dev_zoom_speed is om), but otherwise you get
-- dev_zoom_dir = ‘\x00’ for positive and ‘\x01’ for negative,
-- and dev_zoom_speed will be one less than the magnitude of n.
--
-- Going the other way, I take one of those three direction indicators
-- together with a speed in the range 0 .. 7, and map the pair to
-- −8 .. −8 (8 is symbolized as max_zoom_speed in the code).

proc downcvt_zoom_speed (zoom_speed);
  n := round zoom_speed max −max_zoom_speed min max_zoom_speed;
  case sign n of
  (−1):  return [‘\x01’, −n−1];  -- i.e., abs n − 1
  (0):   return [‘\xff’, om];
  (1):   return [‘\x00’, n−1];
  end case;
end proc;

proc upcvt_zoom_speed (dev_zoom_dir, dev_zoom_speed);
  case dev_zoom_dir of
  (‘\x01’):  return −dev_zoom_speed − 1;
  (‘\xff’):  return 0;
```

```
 ('\x00'): return dev_zoom_speed + 1;
 end case;
end proc;

-- Zoom factor conversions:
--
-- Until I know the relationship between zoom ring position
-- and zoom factor, I will just scale linearly; I might try
-- logarithmically later if it looks like it might work out
-- better.

#define zoom_lo min_zoom
#define zoom_hi max_zoom
#define dev_zoom_lo 0
#define dev_zoom_hi 16#469#

proc downcvt_zoom_factor (zoom_factor);
 u := ((zoom_factor - zoom_lo) /
      (zoom_hi     - zoom_lo)) max 0 min 1;
 -- Rounding is not quite "fair" here but I like it anyway
 return round (u * (dev_zoom_hi - dev_zoom_lo) + dev_zoom_lo);
end proc;

proc upcvt_zoom_factor (dev_zoom_factor);
 u := ((dev_zoom_factor - dev_zoom_lo) /
      (dev_zoom_hi     - dev_zoom_lo)) max 0 min 1;
 return u * (zoom_hi - zoom_lo) + zoom_lo;
end proc;

-- Pan/Tilt rate conversions:
--
-- The device units, degrees per second, seem ideal.  "Down"
-- conversions map your signed pan_speed or tilt_speed to a
-- [sign, magnitude] pair.  The magnitude is suitable for a device
-- Pan/Tilt Speed command, and the sign for Pan/Tilt Start.
--
-- The speed range is −76 to 76 degrees per second for pan, and
-- −70 to 70 for tilt, with positive meaning rightgoing or upgoing
-- respectively.
--
-- “Up” conversions are the obvious inverse of down, without the
-- range check or rounding.

proc downcvt_pan_speed (pan_speed);
  n := round pan_speed max −max_pan_speed min max_pan_speed;
  case sign n of
  (−1):  return [‘\x02’, −n];  -- i.e., abs n
  (0):   return [‘\x00’, 1];
  (1):   return [‘\x01’, n];
  end case;
end proc;

proc upcvt_pan_speed (dev_pan_dir, dev_pan_speed);
  case dev_pan_dir of
  (‘\x02’):  return −dev_pan_speed;
  (‘\x00’):  return 0;
  (‘\x01’):  return dev_pan_speed;
  end case;
end proc;

proc downcvt_tilt_speed (tilt_speed);
  n := round tilt_speed max −max_tilt_speed min max_tilt_speed;
  case sign n of
  (−1):  return [‘\x02’, −n];  -- i.e., abs n
  (0):   return [‘\x00’, 1];
  (1):   return [‘\x01’, n];
  end case;
end proc;

proc upcvt_tilt_speed (dev_tilt_dir, dev_tilt_speed);
  case dev_tilt_dir of
  (‘\x02’):  return −dev_tilt_speed;
  (‘\x00’):  return 0;
  (‘\x01’):  return dev_tilt_speed;
  end case;
end proc;
```

```
-- Pan/Tilt conversions:
--
-- The natural units are degrees.  The Canon takes units of 0.115 deg
-- in both pan and tilt.  Hex 8000 is the midpoint (home), and you can
-- go from hex –30E to 30E in pan, –10A to 0D9 in tilt, representing
-- about –90 to 90 deg in pan, –30 to 25 deg in tilt.
--
-- The device convention for pan is “backwards”.

proc downcvt_pan (pan);
  return 16#8000 + round ((−pan / 0.115) max −16#30E min 16#30E);
end proc;

proc upcvt_pan (dev_pan);
  return (16#8000 − dev_pan) * 0.115;
end proc;

proc downcvt_tilt (tilt);
  return 16#8000 + round ((tilt / 0.115) max −16#10A min 16#0D9);
end proc;

proc upcvt_tilt (dev_tilt);
  return (dev_tilt − 16#8000) * 0.115;
end proc;


-- Trajectory functions

proc max_traj (x);  -- value of ramp or its max_speed extension at x
  assert x ≥ 0;
  n := 1 max ceil ramp_ticks;  -- a point at the ramp end or beyond
  k := ceil x;
  if k < n then
    return accel*k*(x−(k−1)/2);  -- if x=k, this is accel*k*(k+1)/2
  else
    -- k≥n means x is beyond what the ramp covers.  Follow the ramp
    -- to n−1 and then travel at max_speed the rest of the way to x:
```

```
    return traj (x, n−1, max_speed);
  end if;
end proc max_traj;

proc traj (x, k, speed);  -- follow ramp to k, then go maxly to x
  assert is_integer k;
  assert k ≥ 0;
  assert x ≥ k;  -- I don't need this, but else what a strange caller!
  return max_traj (k) + speed∗(x−k);
end proc;

op ms_to_ticks (ms);  -- for example, k and x above are in ticks
  return ms / ms_per_tick;
end op;


-- Endianness-independent packer and unpacker of two-byte spam

op to_two_bytes (i);
  return char (i div 256) + char (i mod 256);
end op;

op from_two_bytes (s);
  return abs s(1) ∗ 256 + abs s(2);
end op;


proc quit_gracefully;
  exit_gracefully ([[str filename seq_fd, seq_fd]]);
end proc;


#include "vc-decode.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.28 vc-mouse.setl

Service provided:
    mouse
Client of service:
    do (vc-do.setl, Section A.11)
Called by parent program:
    vc-toplev.setl (Section A.42)
Textually #includes:
    vc-exit.setl (Section A.15)
    vc-getname.setl (Section A.16)
    vc-msg.setl (Section A.30)
    vc-obtain.setl (Section A.31)
    vc-provide.setl (Section A.32)

Source code:

```
const yhwh = 'vc-mouse.setl';

-- This strange little service is for Java clients that take an
-- unusual view of mouse gestures by doing some local timing and
-- interpretation that result in mouse "events" we agree to call
-- 'click', 'linger', 'jump', 'zoom', and 'stop'.  These are mapped
-- here to combinations of moving, "jumping" (which is just moving
-- without the usual sigmoid speed ramping of the motion trajectory),
-- and zooming (see vc-do.setl).

const width = 320;
const height = 240;
const panlo = −90;
const panhi = 90;
const tiltlo = −30;
const tilthi = 25;

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const server_fd = fileno provide_service ('mouse');

var clients := {};
var do_fd := om;
```

```
loop

 [ready] := select ([{sigterm_fd, server_fd} + domain clients]);

 if sigterm_fd in ready then
   msg (yhwh + ' (' + str pid + ') caught SIGTERM');
   quit_gracefully;
 end if;

 for client = clients(pump_fd) | pump_fd in ready loop
   done_client (pump_fd);
 end loop;

 if server_fd in ready then
   fd := accept (server_fd);
   if fd ≠ om then
     name := getname fd;
     msg (name+' accepted');
     pump_fd := pump();
     if pump_fd = −1 then
       -- child
       do_fd := fileno obtain_service ('do');
       loop
         if (line := getline fd) ≠ om and
            #(t := split (line)) ≥ 1 then
           case
           when t(1) = 'click'
                and #t = 3
                and is_num t(2)
                and is_num t(3)  ⇒
             x := val t(2);
             y := val t(3);
             pan_norm := (x − width/2) / (width/2);
             tilt_norm := (height/2 − y) / (height/2);
             zoom := do_get ('zoom_factor');
             dist_norm := sqrt (pan_norm**2 + tilt_norm**2);
             zoom_scale := 1.618 ** (2 − 4*dist_norm);
```

```
  dpan := pan_norm * 40 / zoom;
  dtilt := tilt_norm * 30 / zoom;
  if zoom_scale ≤ 1 then
    do_zoom_by (zoom_scale);
    do_move_by (dpan, dtilt);
  else
    do_move_by (dpan, dtilt);
    do_zoom_by (zoom_scale);
  end if;
  printa (fd); -- reply with empty line
when t(1) = 'linger'
    and #t = 3
    and is_num t(2)
    and is_num t(3) ⇒
  x := val t(2);
  y := val t(3);
  pan_norm := (x − width/2) / (width/2);
  tilt_norm := (height/2 − y) / (height/2);
  zoom := do_get ('zoom_factor');
  pan_rate  := sign pan_norm  * pan_norm**2 * 60 / zoom;
  tilt_rate := sign tilt_norm * tilt_norm**2 * 60 / zoom;
  do_move_speed (pan_rate, tilt_rate);
  do_move_start;
  printa (fd); -- reply with empty line
when t(1) = 'jump'
    and #t = 3
    and is_num t(2)
    and is_num t(3) ⇒
  -- this command uses the "natural" units
  pan := val t(2);
  tilt := val t(3);
  do_jump_to (pan, tilt);
  printa (fd); -- reply with empty line
when t(1) = 'zoom'
    and #t = 2
    and is_num t(2) ⇒
  zoom := val t(2);
  do_zoom_to (zoom max 1 min 10);
```

```
          printa (fd);  -- reply with empty line
        when t(1) = 'stop'
            and #t = 1  ⇒
          do_move_stop;
          printa (fd);  -- reply with empty line
        otherwise  ⇒
          stop;
        end case;
      else
        stop;
      end if;
    end loop;
    assert false;
  end if;
  -- parent continues here
  close (fd);
  client := {};
  client.name := name;
  clients(pump_fd) := client;
 end if;
end if;

end loop;

proc new_cmd (name);
 cmd := {};
 cmd.name := name;
 return cmd;
end proc;

proc do_cmd (cmd);
 writea (do_fd, cmd);
 geta (do_fd, response_line);
 return response_line;  --- currently with no check
end proc;

proc do_jump_to (pan, tilt);
 do_jump ('To', pan, tilt);
```

```
end proc;

proc do_jump_by(pan, tilt);
  do_jump ('By', pan, tilt);
end proc;

proc do_jump (toby, pan, tilt);
  cmd := new_cmd ('Jump');
  cmd.subcmd := toby;
  cmd.pan := pan;
  cmd.tilt := tilt;
  do_cmd (cmd);
end proc;

proc do_move_to (pan, tilt);
  do_move ('To', pan, tilt);
end proc;

proc do_move_by (pan, tilt);
  do_move ('By', pan, tilt);
end proc;

proc do_move (toby, pan, tilt);
  cmd := new_cmd ('Move');
  cmd.subcmd := toby;
  cmd.pan := pan;
  cmd.tilt := tilt;
  do_cmd (cmd);
end proc;

proc do_move_start;
  cmd := new_cmd ('Move');
  cmd.subcmd := 'Start';
  do_cmd (cmd);
end proc;

proc do_move_stop;
  cmd := new_cmd ('Move');
  cmd.subcmd := 'Stop';
  do_cmd (cmd);
end proc;

proc do_move_speed (pan_speed, tilt_speed);
  cmd := new_cmd ('Move');
  cmd.subcmd := 'Speed';
  cmd.pan_speed := pan_speed;
  cmd.tilt_speed := tilt_speed;
  do_cmd (cmd);
end proc;

proc do_zoom_to (factor);
  cmd := new_cmd ('Zoom');
  cmd.subcmd := 'To';
  cmd.zoom_factor := factor;
  do_cmd (cmd);
end proc;

proc do_zoom_by (scale);
  cmd := new_cmd ('Zoom');
  cmd.subcmd := 'By';
  cmd.zoom_scale := scale;
  do_cmd (cmd);
end proc;

proc do_get (what);
  cmd := new_cmd ('Get');
  cmd.what := what;
  return unstr do_cmd (cmd);
end proc;

op is_num (a);
  return a('^[+-]?[0-9]+(\\.[0-9]+)?$') ≠ om;
end op;

proc done_client (pump_fd);
  msg (clients(pump_fd).name + ' done');
  close (pump_fd);
  clients(pump_fd) := om;
end proc done_client;

proc quit_gracefully;
  exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                                        client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.29 vc-mover.setl

Service provided:
  mover
Client of service:
  do (vc-do.setl, Section A.11)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-simpler.setl (Section A.40)

Source code:

```
const yhwh = 'vc-mover.setl';

-- Simplified "Move To" command interface.

-- The client just sends a pair of numbers (pan and tilt) on each line.

-- The name of the service we provide:
#define service_name 'mover'

-- The number of parameters on the command:
#define n_parms  2

-- The full details of the command we will send to the do service:
#define build_cmd  \
 cmd := {};  \
 cmd.name := 'Move';  \
 cmd.subcmd := 'To';  \
 cmd.pan := val t(1);  \
 cmd.tilt := val t(2);

#include "vc-simpler.setl"
```

A.30 vc-msg.setl

Textually #included by:

vc-camera.setl (Section A.4)
vc-do.setl (Section A.11)
vc-event.setl (Section A.12)
vc-giver.setl (Section A.17)
vc-httpd.setl (Section A.19)
vc-image.setl (Section A.20)
vc-input.setl (Section A.22)
vc-javent.setl (Section A.23)
vc-model.setl (Section A.27)
vc-mouse.setl (Section A.28)
vc-ptz.setl (Section A.33)
vc-push.setl (Section A.34)
vc-recv.setl (Section A.36)
vc-send.setl (Section A.38)
vc-seq.setl (Section A.39)
vc-simpler.setl (Section A.40)
vc-snap.setl (Section A.41)

Source code:

```
-- If the program which #includes this file is a server started by
-- vc-toplev.setl (or a subprocess thereof), and identifies itself in
-- the string yhwh, then this routine spews a message that will end
-- up in the log file, because vc-toplev.setl captures all output sent
-- to stderr by its subprocesses and feeds that output through to the
-- spew routine of vc-admin.setl, prefixed by a timestamp and yhwh.

proc msg (s);
  printa (stderr, yhwh, ':', s);
end proc;
```

A.31 vc-obtain.setl

Textually #included by:

- vc-do.setl (Section A.11)
- vc-giver.setl (Section A.17)
- vc-httpd.setl (Section A.19)
- vc-javent.setl (Section A.23)
- vc-mouse.setl (Section A.28)
- vc-ptz.setl (Section A.33)
- vc-push.setl (Section A.34)
- vc-seq.setl (Section A.39)
- vc-simpler.setl (Section A.40)
- vc-snap.setl (Section A.41)

Source code:

```
-- Open a TCP client port on a named service:
proc obtain_service (serv_name);
 var serv_host, serv_port;
 [serv_host, serv_port] := find_service (serv_name);
 return open (serv_host + ':' + str serv_port, 'socket');
end proc;

-- Find the location of a service, given its registered name:
proc find_service (serv_name);
 var fd, serv_info;
 fd := fileno open (getenv 'VC_LOOKUP', 'socket');
 writea (fd, serv_name);
 reada (fd, serv_info);
 close (fd);
 return serv_info;
end proc;
```

A.32 vc-provide.setl

Textually #included by:

- vc-camera.setl (Section A.4)
- vc-do.setl (Section A.11)
- vc-event.setl (Section A.12)
- vc-giver.setl (Section A.17)
- vc-httpd.setl (Section A.19)
- vc-image.setl (Section A.20)
- vc-javent.setl (Section A.23)
- vc-mouse.setl (Section A.28)
- vc-push.setl (Section A.34)
- vc-simpler.setl (Section A.40)
- vc-snap.setl (Section A.41)

Source code:

```
-- Open a TCP server port and publish its availability.  The port
-- number will be chosen arbitrarily if not given in portnum:
proc provide_service (serv_name, portnum(*));
  const sock = open (str (portnum(1) ? 0), 'server-socket');
  if sock ≠ om then
    -- Use hostname in place of 'localhost' on a distributed system:
    publish_service (serv_name, 'localhost', port sock, pid);
  end if;
  -- If om for non-zero portnum, client may wish to wait and retry:
  return sock;
end proc;

-- Publish the availability of a service by registering its name and
-- location:
proc publish_service (serv_name, serv_host, serv_port, serv_pid);
  const fd = fileno open (getenv 'VC_PUBLISH', 'socket');
  writea (fd, serv_name, [serv_host, serv_port, serv_pid]);
  close (fd);
  -- Redundant with the environment variable, for external parties:
  putfile ('vc-tcp/'+serv_name, serv_host + ':' + str serv_port);
end proc;
```

A.33 vc-ptz.setl

Client of services:
    do (vc-do.setl, Section A.11)
    notice (vc-do.setl, Section A.11)
Called by parent program:
    vc-camera.setl (Section A.4)
Textually #includes:
    vc-decode.setl (Section A.10)
    vc-exit.setl (Section A.15)
    vc-getname.setl (Section A.16)
    vc-msg.setl (Section A.30)
    vc-obtain.setl (Section A.31)

Source code:

```
const yhwh = 'vc-ptz.setl';

-- This program is instantiated as a pumping co-process child of
-- vc-camera.setl for each client that is using the high-level
-- command interface to the Canon VC-C3 pan/tilt/zoom (ptz) camera
-- controller.
--
-- The file descriptor of the connected client's socket, which
-- this program inherits, is identified on the program invocation
-- command line.

const arg_fd = fileno open (val command_line(1), 'socket');
const in_fd = arg_fd,
    out_fd = arg_fd;  -- mnemonic references to the inherited socket

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals

const do_fd = open_do_server();  -- mid-level command (do) server
var notice_fd := om;  -- mid-level event notification service

open ('SIGPIPE', 'ignore');  -- retain control on EPIPE output errors

p ('Welcome to the Canon VC-C3 pan/tilt/zoom camera control server.');
```

```
p ('Type Help for help.  Cavil and whine to dB (bacon@cs.nyu.edu).');
out ('.');  -- "end of help" marker

prev_words := ['Help'];

loop

 pool := {sigterm_fd, in_fd, notice_fd};  -- notice_fd may be om
 [ready] := select ([pool]);

 if sigterm_fd in ready then
  msg (yhwh + ' (' + str pid + ') caught SIGTERM');
  quit_gracefully;
 end if;

 if in_fd in ready then
  if (line := getline in_fd) = om then
   quit_gracefully;
  end if;
  words := split (line);
  if #words = 0 then
   words := prev_words;
  else
   prev_words := words;
  end if;
  cmd := words(1);
  case of

--- This command language needs a comment convention!  How about "#"?

  (cmd ceq 'Help'):  -- Help [command-name]
   -- Send lines prefixed with ">", followed by a line containing
   -- a single "."
   if #words = 1 then
    p ('');
    p ('Commands are:');
    p ('');
    p ('Help [command-name]');
    p ('Mode {Host | RC}');
    p ('Notify {On | Off}');
    p ('Zoom {[To] factor | By factor | In | Out} [At speed]');
    p ('Move {[To] pan tilt | By pan tilt} [[In] ms | At speed]');
    p ('{Up | Down | Left | Right} deg [[In] ms | At speed]');
    --- still to document:  Jump
    p ('Ramp ms');
    p ('Show {Mode | Notify | Zoom | Move | Position | Ramp}');
    p ('Clear');
    p ('Reload');
    p ('Setup');
    p ('Reset');
    p ('Check');
    p ('Quit');
    p ('');
    p ('A null command (empty line) repeats the previous command.');
  else
    cmd_name := words(2);
    case of

    (cmd_name ceq 'Help'):
      p ('');
      p ('Help');
      q ('Gives a compact synopsis of all commands, with optional'+
        ' words shown in brackets [ ], grouping indicated by'+
        ' braces { }, and alternatives separated by bars |.');
      q ('All command names and arguments are case-insensitive,'+
        ' though for clarity they are shown here as literal names'+
        ' starting with an uppercase letter.  Substitute a value'+
        ' for any (possibly hyphenated) name that begins with a'+
        ' lowercase letter.  Numbers may include signs and decimal'+
        ' points.');
      q ('Help is the only command besides Show which produces'+
        ' output back to you, the client, when asynchronous'+
        ' notification is off (see the Notify command).  You can'+
        ' tell where a piece of help ends by where the ">" lines'+
        ' leave off and the final "." on a line by itself occurs. '+
        ' Server usage errors (your protocol mistakes) are also'+
        ‘ reported in this "help" format.  Output from Show always’+
        ‘ consists of a single line, as does each asynchronous’+
        ‘ notification (event message), so their ends are also’+
        ‘ easy to recognize.’);
      p (‘’);
      p (‘Help command-name’);
      q (‘Tells you all about a specific command.’);

    (cmd_name ceq ‘Notify’):
      p (‘’);
      p (‘Notify On’);
      q (‘Turns on asynchronous notification.  You (the client)’+
        ‘ will get an event message, formatted as a command’+
        ‘ recognized by this server for convenience in playback,’+
        ‘ whenever there is a change in the mode, zoom, pan/tilt,’+
        ‘ or ramp, and whenever a zoom or pan/tilt limit is’+
        ‘ reached. ’+
        ‘ [Other messages, with no corresponding command but’+
        ‘ formatted similarly, will later be added.  For now,’+
        ‘ there is a catch-all message "Canon", showing things the’+
        ‘ hardware is saying.]’);
      p (‘’);
      p (‘Notify Off’);
      q (‘Turns off asynchronous notification.  You can still get’+
        ‘ information synchronously by using the Show command.’);

    (cmd_name ceq ‘Zoom’):
      p (‘’);
      p (‘Note on zoom speeds:’);
      q (‘The Canon has 8 speeds going in (TELE) and out (WIDE),’+
        ‘ and accordingly the speed given in any [At speed] clauses’+
        ‘ on Zoom commands should be a number in the range 1 to 8. ’+
        ‘ Zooming in all the way from a zoom factor of 1 to a zoom’+
        ‘ factor of 10, or the reverse, seems to take about 2’+
        ‘ seconds at the maximum speed of 8, and about 9 seconds’+
        ‘ at the minimum speed of 1.’);
      p (‘Zoom [To] factor [At speed]’);
      q (‘Sets the current zoom factor to a floating-point value’+
        ‘ between 1 and 10.’);
      p (‘Zoom By factor [At speed]’);
      q (‘Scales the current zoom factor by the specified factor.’);
      p (‘Zoom In [At speed]’);
      q (‘Equivalent to Zoom By 1.618.’);
      p (‘Zoom Out [At speed]’);
      q (‘Equivalent to Zoom By 0.618.’);

    (cmd_name ceq ‘Move’):
      p (‘’);
      p (‘Move [To] pan tilt [[In] ms] | At speed]’);
      q (‘Points the camera at pan degrees azimuth, tilt degrees’+
        ‘ elevation, and stores these as the current values.’);
      q (‘Positive means right for pan, up for tilt.’);
      q (‘Range is -90 to 90 for pan, -30 to 25 for tilt.’);
      q (‘Resolution is 0.115 deg.’);
      q (‘The angular trajectory is shaped at each end by the’+
        ‘ parabola suggested by the Ramp period.  If the angular’+
        ‘ distance to move is large enough, maximum speed will be’+
        ‘ sustained in the interval between the acceleration and’+
        ‘ deceleration ramps unless constrained by the optional In’+
        ‘ or At specification.’);
      q (‘If "[In] ms" is specified, the server will try to plan a’+
        ‘ camera motion trajectory that takes ms milliseconds.’);
      q (‘If instead "At speed" is specified, the trajectory speed’+
        ‘ will be limited to the given maximum during the’+
        ‘ constant-speed interval between acceleration and’+
        ‘ deceleration ramps.’);
      q (‘The units of speed in "At speed" are deg/sec, with a’+
        ‘ resolution of 1 deg/sec and a range of 1 to 70 deg/sec.’);
      p (‘Move By pan tilt [[In] ms] | At speed]’);
      q (‘Adds pan degrees azimuth and tilt degrees elevation to’+
        ‘ the current pan and tilt values, and calls Move [To].’);

    (cmd_name ceq ‘Up’):
      p (‘’);
      p (‘Up deg [[In] ms] | At speed]’);
      q (‘Synonymous with "Move By 0 deg" plus In/At options.’);
```

```
(cmd_name ceq 'Down'):
  p ('');
  p ('Down deg [[In] ms] | At speed]');
  q ('Synonymous with "Move By 0 -deg" plus In/At options.');

(cmd_name ceq 'Left'):
  p ('');
  p ('Left deg [[In] ms] | At speed]');
  q ('Synonymous with "Move By -deg 0" plus In/At options.');

(cmd_name ceq 'Right'):
  p ('');
  p ('Right deg [[In] ms] | At speed]');
  q ('Synonymous with "Move By deg 0" plus In/At options.');

(cmd_name ceq 'Ramp'):
  p ('');
  p ('Ramp ms');
  q ('Sets the number of milliseconds the pan/tilt apparatus'+
    ' will take to get up to maximum speed on Move [To] and'+
    ' Move By requests, and also how long it will take to slow'+
    ' down as the destination is approached.');
  q ("The default is 500 ms, which shouldn't jerk the platform"+
    ' very violently.  Should look way smooth, too, eh.');

(cmd_name ceq 'Show'):
  p ('');
  p ('All Show commands produce their output in the form of a');
  p ('command that could later be fed back in to the server to');
  p ('re-establish the state reported by the Show.');
  p ('');
  p ('Show Mode');
  q ('Yields Mode Host or Mode RC.');
  p ('Show Notify');
  q ('Yields Notify On or Notify Off.');
  q ('Each asynchronous notification (event message) and Show'+
    ' result is sent to you, the client, on a single,'+
    ‘ newline-terminated line.’);
  p (‘Show Zoom’);
  q (‘Yields the current zoom factor as a Zoom [To] command.’);
  p (‘Show {Position | Move}’);
  q (‘Yields the current pan and tilt angles as a Move [To]’+
     ‘ command.’);
  p (‘Show Ramp’);
  q (‘Yields a Ramp command for the current ramp period.’);
  p (‘’);
  p (‘See also Reload and Check.’);

(cmd_name ceq ‘Mode’):
  p (‘’);
  p (‘Mode Host’);
  q (‘Puts the pan/tilt and zoom apparatus into a state where’+
    ‘ it is receptive to commands from the computer rather’+
    ‘ than from the hand-held remote control.’);
  q (‘(1) Uses Clear to toggle the RTS line down and up, (2)’+
    ‘ requests the device mode change, and (3) calls Setup’+
    ‘ for auto-detection of the home position and for server’+
    ‘ state refreshment.’);
  p (‘’);
  p (‘Mode RC’);
  q (‘Zoom and Move commands from the computer (meaning you,’+
    ‘ the client!) will be ignored until the next switch to’+
    ‘ Mode Host.’);
  q (‘(1) Uses Clear to toggle the RTS line down and up, and’+
    ‘ (2) requests the device mode change.’);

(cmd_name ceq ‘Clear’):
  p (‘’);
  p (‘Clear’);
  q (‘Toggles the RTS line of the RS-232 communications link’+
    ‘ down and up in an effort to cancel a "wait state" entered’+
    ‘ by the Canon VC-C3.’);
  q (‘Merely wastes a little time if the hardware is not in’+
    ‘ such a state.’);
  q (‘Called automatically by the Mode command, which is called’+
    ‘ by Reset.’);

(cmd_name ceq ‘Reload’):
  p (‘’);
  p (‘Reload’);
  q (‘Causes this server to refresh its record of the hardware’+
    ‘ state by reading the current zoom and pan/tilt parameters’+
    ‘ from the Canon VC-C3.’);
  q (‘If asynchronous notification is on, sends you Zoom and’+
    ‘ Move event messages reflecting the newly read values.’);
  q (‘Try this command if you think Show might be lying to’+
    ‘ you.  See also Check.’);
  q (‘Reload is called automatically by Setup, which is called’+
    ‘ by Mode Host, which is called by Reset in host mode.’);
  q (‘Reload cannot detect the control mode (Host or RC).  ’+
    ‘ Hence this parameter is meekly assumed to be unchanged,’+
    ‘ and the best you can do for definiteness is to set it’+
    ‘ using the Mode command.’);

(cmd_name ceq ‘Setup’):
  p (‘’);
  p (‘Setup’);
  q (‘Causes the Canon VC-C3 hardware to auto-detect the’+
    ‘ pan/tilt "home" position, and then calls Reload to’+
    “ refresh the server′s record of the hardware state.”);
  q (‘Done automatically as the final stage of Mode Host,’+
    ‘ which is called by Reset in host mode.’);
  q (‘Can take as long as 4 seconds, as the camera swings’+
    ‘ wildly to the home position and then back to wherever it’+
    ‘ was.’);

(cmd_name ceq ‘Reset’):
  p (‘’);
  p (‘Reset’);
  q (‘Causes the Canon VC-C3 camera control unit (CCU) to’+
    ‘ re-initialize.  The CCU will lower the CTS line of the’+
    ‘ RS-232 communications link for 3.8 seconds during this’+
    ‘ process.  When CTS is later raised, the server will’+
     ' attempt to establish the control mode that obtained'+
     ' before the Reset, by calling the Mode command.');

(cmd_name ceq 'Check'):
  p ('');
  p ('Check');
  q ('Perform a "sanity check" on the Canon VC-C3 camera'+
     ' control unit (CCU), and do a Reset if it appears to be'+
     ' malfunctioning.  Sometimes, for reasons unknown, the CCU'+
     ' gets into a state where it responds normally on the'+
     ' serial line but is in fact ignoring zoom and/or motion'+
     ' commands; Check effectively does a Show before and after'+
     ' a Reload, and looks for discrepancies.  It is primarily'+
     ' intended for the use of external automata, but if you'+
     ' find the CCU not responding, you can see Check in action'+
     ' by zooming (say) to a setting far from what Show is'+
     ' reporting, doing a Check, and watching for the effects'+
     ' of a Reset (e.g., transient camera motion, image'+
     ' blanking, and, if you have Notify On, the "event"'+
     ' messages that normally result from a Reset).');

(cmd_name ceq 'Quit'):
  p ('');
  p ('Quit');
  q ('Asks the server to drop the network connection.');
  q ('For technical reasons, it is mildly preferable that you,'+
     ' the client, drop the network connection first, usually'+
     ' simply by closing it.  The server will follow suit. '+
     ' This avoids the compulsory 2MSL wait to retire the'+
     " half-association in the TCP module on the server's"+
     ' host when the server drops first.  So in other words,'+
     " unless you have some good reason to, DON'T USE Quit,"+
     ' but merely close your socket when you are done with me. '+
     ' It is really no big deal, though.');

else
  p ('');
  p ('Invalid argument to Help command - try Help Help.');
```

```
    end case;

  end if;
  out ('.');  -- "end of help" marker

 (cmd ceq 'Notify'):  -- Notify {On | Off}
  if #words = 2 then
    switch := words(2);
    case of
    (switch ceq 'On'):
      notice_fd ?:= open_notice_server();
    (switch ceq 'Off'):
      if notice_fd ≠ om then
        close (notice_fd);
        notice_fd := om;
      end if;
    else
      help ('Notify argument must be On or Off – try Help Notify');
    end case;
  else
    help ('Notify command requires 1 argument – try Help Notify');
  end if;

 (cmd ceq 'Zoom'):  -- Zoom {Start | Stop}
                  -- Zoom Speed zoom-speed
                  -- Zoom {[To] factor | By factor | In | Out}
                  --                    [At speed]
  case of
  ((#words = 2) and (words(2) ceq 'Start')):
    do_zoom ('Start');
  ((#words = 2) and (words(2) ceq 'Stop')):
    do_zoom ('Stop');
  ((#words = 3) and (words(2) ceq 'Speed')
              and is_num words(3)):
    zoom_speed := val words(3);
    do_zoom ('Speed', zoom_speed);
  else
    if #words ≥ 2 then
```

```
          words(1 .. 1) := [ ];
          if words(1) ceq 'To' then
            if #words ≥ 2 and is_num words(2) then
              do_fussy_zoom ('To', val words(2), words(3 .. ));
            else
              help ('Zoom command parameter error - try Help Zoom');
            end if;
          elseif is_num words(1) then
            do_fussy_zoom ('To', val words(1), words(2 .. ));
          elseif words(1) ceq 'By' then
            if #words ≥ 2 and is_num words(2) then
              do_fussy_zoom ('By', val words(2), words(3 .. ));
            else
              help ('Zoom command parameter error - try Help Zoom');
            end if;
          elseif words(1) ceq 'In' then
            do_fussy_zoom ('By', 1.618, words(2 .. ));
          elseif words(1) ceq 'Out' then
            do_fussy_zoom ('By', 0.618, words(2 .. ));
          else
            help ('Zoom command parameter error - try Help Zoom');
          end if;
        else
          help ('Zoom command parameter error - try Help Zoom');
        end if;
      end case;

    --- The Start, Stop, and Speed subcommands are deprecated and
    --- de-documented, and may disappear from a future release of
    --- the software:

    (cmd ceq 'Move'):   -- Move {Start | Stop}
                        -- Move Speed pan-speed tilt-speed
                        -- Move {[To] pan tilt | By pan tilt}
                        --                [[In] ms | At speed]
      case of
      ((#words = 2) and (words(2) ceq 'Start')):
        do_move ('Start');
```

```
   ((#words = 2) and (words(2) ceq 'Stop')):
     do_move ('Stop');
   ((#words = 4) and (words(2) ceq 'Speed')
               and is_num words(3)
               and is_num words(4)):
     pan_speed := val words(3);
     tilt_speed := val words(4);
     do_move ('Speed', pan_speed, tilt_speed);
   else
     if #words ≥ 3 then
       words(1 .. 1) := [ ];
       toby := 'To';
       if words(1) ceq 'To' then
         words(1 .. 1) := [ ];
       elseif words(1) ceq 'By' then
         words(1 .. 1) := [ ];
         toby := 'By';
       end if;
       if is_num words(1) and
          is_num words(2) then
         pan := val words(1);
         tilt := val words(2);
         do_fussy_move (toby, pan, tilt, words(3 .. ));
       else
         help ('Move command parameter error - try Help Move');
       end if;
     else
       help ('Move command parameter error - try Help Move');
     end if;
   end case;

 (cmd ceq 'Speed'):  -- Speed pan-speed tilt-speed
   if #words = 3 and is_num words(2)
               and is_num words(3) then
     pan_speed := val words(2);
     tilt_speed := val words(3);
     do_move ('Speed', pan_speed, tilt_speed);
   else
```

```
    help ('Speed command parameter error');
   end if;

  (cmd ceq 'Up'):  -- Up deg [[In] ms | At speed]
   if #words ≥ 2 and is_num words(2) then
    tilt := val words(2);
    do_fussy_move ('By', 0, tilt, words(3..));
   else
    help ('Up command parameter error – try Help Up');
   end if;

  (cmd ceq 'Down'):  -- Down deg [[In] ms | At speed]
   if #words ≥ 2 and is_num words(2) then
    tilt := val words(2);
    do_fussy_move ('By', 0, −tilt, words(3..));
   else
    help ('Down command parameter error – try Help Down');
   end if;

  (cmd ceq 'Left'):  -- Left deg [[In] ms | At speed]
   if #words ≥ 2 and is_num words(2) then
    pan := val words(2);
    do_fussy_move ('By', −pan, 0, words(3..));
   else
    help ('Left command parameter error – try Help Left');
   end if;

  (cmd ceq 'Right'):  -- Right deg [[In] ms | At speed]
   if #words ≥ 2 and is_num words(2) then
    pan := val words(2);
    do_fussy_move ('By', pan, 0, words(3..));
   else
    help ('Right command parameter error – try Help Right');
   end if;

  (cmd ceq 'Jump'):
   if #words = 3 and is_num words(2) and
                is_num words(3) then
    pan := val words(2);
    tilt := val words(3);
    do_jump ('To', pan, tilt);
  else
    help ('Jump command parameter error');  --- later "try Help Jump"
  end if;
  --- still to implement:  "Jump By"

(cmd ceq 'Ramp'):  -- Ramp ms
  if #words = 2 and is_num words(2) then
    ms := val words(2);
    do_ramp (ms);
  else
    help ('Ramp command requires 1 numeric argument – try Help Ramp');
  end if;

(cmd ceq 'Show'):  -- Show {Mode | Notify |
                   --       Zoom |
                   --       Position | Move |
                   --       Ramp}
  case of
  (words(2 .. ) ceq ['Mode']):
    which_mode := do_get ('mode');
    out ('Mode ' + which_mode);
  (words(2 .. ) ceq ['Notify']):
    out ('Notify ' + if notice_fd ≠ om then 'On' else 'Off' end);
  (words(2 .. ) ceq ['Zoom']):
    zoom_factor := do_get ('zoom_factor');
    out ('Zoom ' + fixed (zoom_factor, 0, 3));
  (words(2 .. ) ceq ['Zooming']):
    zooming := do_get ('zooming');
    out ('Zoom ' + if zooming then 'Start' else 'Stop' end);
  (words(2 .. ) ceq ['Zoom','Speed']):
    zoom_speed := do_get ('zoom_speed');
    out ('Zoom Speed ' + fixed (zoom_speed, 0, 1));
  (words(2 .. ) ceq ['Position'],
   words(2 .. ) ceq ['Move']):
    [pan, tilt] := do_get ('position');
    out ('Move To ' + fixed (pan, 0, 3) + ' ' + fixed (tilt, 0, 3));
  (words(2 .. ) ceq ['Moving']):
    moving := do_get ('moving');
    out ('Move ' + if moving then 'Start' else 'Stop' end);
  (words(2 .. ) ceq ['Speed'],
   words(2 .. ) ceq ['Move','Speed']):
    [pan_speed, tilt_speed] := do_get ('move_speed');
    out ('Move Speed  ' + fixed (pan_speed, 0, 1) +
                ' ' + fixed (tilt_speed, 0, 1));
  (words(2 .. ) ceq ['Ramp']):
    ms := do_get ('ramp');
    out ('Ramp ' + str ms);
  else
    help ('Show command parameter error - try Help Show');
  end case;

(cmd ceq 'Mode'):  -- Mode {Host | RC}
  if #words = 2 then
    which_mode := words(2);
    case of
    (which_mode ceq 'Host'):
      do_mode ('Host');  -- Canon under computer control
    (which_mode ceq 'RC'):
      do_mode ('RC');  -- Canon under zapper control
    else
      help ('Unrecognized mode "'+which_mode+'" - try Help Mode');
    end case;
  else
    help ('Mode command requires 1 argument - try Help Mode');
  end if;

(cmd ceq 'Clear'):  -- Clear
  if #words = 1 then
    do_clear;
  else
    help ('Clear command takes no arguments - try Help Clear');
  end if;
```

```
(cmd ceq 'Reload'):  -- Reload
 if #words = 1 then
  do_reload;
 else
  help ('Reload command takes no arguments – try Help Reload');
 end if;

(cmd ceq 'Setup'):  -- Setup
 if #words = 1 then
  do_setup;
 else
  help ('Setup command takes no arguments – try Help Setup');
 end if;

(cmd ceq 'Reset'):  -- Reset
 if #words = 1 then
  do_reset;
 else
  help ('Reset command takes no arguments – try Help Reset');
 end if;

(cmd ceq 'Check'):  -- Check
 if #words = 1 then
  do_check;
 else
  help ('Check command takes no arguments – try Help Check');
 end if;

(cmd ceq 'Hex'):  -- Hex cmd
 -- This command is not advertised by Help.
 if #words = 2 and is_hex words(2) then
  do_hex (words(2));
 else
  help ('Hex command requires 1 hex argument');
 end if;

(cmd ceq 'Quit'):  -- Quit
 if #words = 1 then
```

```
      quit_gracefully;
    else
      help ('Quit command takes no arguments - try Help Quit');
    end if;
  else
    help ('Unrecognized command - try Help');
  end case;

 end if in_fd;

 if notice_fd $\neq$ om and notice_fd in ready then
   reada (notice_fd, notice);
   if eof then
     help ('Mid-level notice service crashed - sorry.');
     msg ('EOF from '+filename notice_fd+' - closing');
     close (notice_fd);
     notice_fd := om;
   else
     if is_string notice then
       out ('Canon '+notice);
     else
       out ('Canon '+str notice);
     end if;
   end if;
 end if notice_fd;

end loop;

-- Case-insensitive comparison of strings or aggregates thereof
op ceq (a, b);
 return if  is_string a  then  to_upper a = to_upper b
        else   #a = #b  and   forall s = a(i) | s ceq b(i)
        end if;
end op;

op is_num (a);
 return a('^[+-]?[0-9]+(\\.[0-9]+)?$') $\neq$ om;
end op;
```

```
op is_hex (s);
  return is_string s and #s mod 2 = 0 and s('^[0-9a-fA-F]*$') = s;
end op;

proc out (s);
  --
  -- We take the trouble to make sure there is a '\r' (carriage return)
  -- before each '\n' (newline), because that is strictly speaking how
  -- line-oriented Internet programs are supposed to communicate, and if
  -- the client of this program happens to be telnet running under
  -- DOS/Windows, these carriage-return characters will be very welcome.
  -- Conversely, they will not hurt such clients running in (say) xterms
  -- under Unix.
  --
  -- There is no check for output errors here, because it is fair to
  -- assume that if the client drops the connection, this will be seen
  -- soon enough on the input side (as an EOF).
  --
  printa (out_fd, s+'\r');
end proc;

proc p (s);  -- spew a line of a help message
  out ('>'+s);
end proc;

proc q (s);  -- fill and spew a point-paragraph
  para := filter ('fmt -60', s);
  mash := ' - ';
  for line in split (para(1 .. #para-1), '\n') loop
    p (mash+line);
    mash := '   ';
  end loop;
end proc;

proc help (s);  -- spew a diagnostic in the form of a help message
  p (s);
  out ('.');  -- "end of help" marker
```

```
end proc;

proc new_cmd (name);
 cmd := {};
 cmd.name := name;
 return cmd;
end proc;

proc do_fussy_zoom (toby, zoom, words);
 if #words = 2 and words(1) ceq 'At'
              and is_num words(2) then
   speed := val words(2);
   do_zoom (toby, zoom, speed);
 elseif #words = 0 then
   do_zoom (toby, zoom);
 else
   help ('Error in [At speed] option - try Help Zoom');
 end if;
end proc;

proc do_zoom (subcmd, x(*));
 cmd := new_cmd ('Zoom');
 cmd.subcmd := subcmd;
 case subcmd of
 ('Start'):  pass;
 ('Stop'):   pass;
 ('Speed'):  [cmd.zoom_speed,  -        ] := x;
 ('To'):     [cmd.zoom_factor, cmd.speed] := x;
 ('By'):     [cmd.zoom_scale,  cmd.speed] := x;
 end case;
 do_cmd (cmd);
end proc;

proc do_fussy_move (toby, pan, tilt, words);
 if #words = 1 and is_num words(1) or
    #words = 2 and words(1) ceq 'In'
              and is_num words(2) then
   ms := val words(#words);
```

```
    do_move (toby, pan, tilt, 'In', ms);
  elseif #words = 2 and words(1) ceq 'At'
                  and is_num words(2) then
    speed := val words(#words);
    do_move (toby, pan, tilt, 'At', speed);
  elseif #words = 0 then
    do_move (toby, pan, tilt);
  else
    help ('Error in In or At option - try Help Move');
  end if;
end proc;

proc do_move (subcmd, x(*));
  cmd := new_cmd ('Move');
  cmd.subcmd := subcmd;
  case subcmd of
  ('Start'):  pass;
  ('Stop'):   pass;
  ('Speed'):  [cmd.pan_speed, cmd.tilt_speed] := x;
  ('To','By'):
    [cmd.pan, cmd.tilt] := x;
    if x(3) = 'In' then
      cmd.ms := x(4);
    elseif x(3) = 'At' then
      cmd.speed := x(4);
    end if;
  end case;
  do_cmd (cmd);
end proc;

proc do_jump (toby, pan, tilt);
  cmd := new_cmd ('Jump');
  cmd.subcmd := toby;
  cmd.pan := pan;
  cmd.tilt := tilt;
  do_cmd (cmd);
end proc;
```

```
proc do_ramp (ms);
  cmd := new_cmd ('Ramp');
  cmd.ms := ms;
  do_cmd (cmd);
end proc;

proc do_mode (which_mode);
  cmd := new_cmd ('Mode');
  cmd('mode') := which_mode;
  do_cmd (cmd);
end proc;

proc do_clear;
  cmd := new_cmd ('Clear');
  do_cmd (cmd);
end proc;

proc do_reload;
  cmd := new_cmd ('Reload');
  do_cmd (cmd);
end proc;

proc do_setup;
  cmd := new_cmd ('Setup');
  do_cmd (cmd);
end proc;

proc do_reset;
  cmd := new_cmd ('Reset');
  do_cmd (cmd);
end proc;

proc do_check;
  cmd := new_cmd ('Check');
  do_cmd (cmd);
end proc;

proc do_hex (device_cmd);
```

```
  cmd := new_cmd ('Hex');
  cmd.cmd := device_cmd;
  do_cmd (cmd);
end proc;

proc do_get (what);
  cmd := new_cmd ('Get');
  cmd.what := what;
  value := unstr do_cmd (cmd);
  return value;
end proc;

proc do_cmd (cmd);
  writea(do_fd, cmd);
  geta (do_fd, response_line);
  if eof then
    help ('Mid-level command service crashed - sorry.');
    msg ('EOF from '+filename do_fd+' - closing');
    quit_gracefully;  -- we can't do much without the do service
  end if;
  return response_line;
end proc;

proc open_do_server();
  fd := obtain_service ('do');
  if fd = om then
    help ('Mid-level command service down - sorry.');
    msg ('could not open "do" service');
    quit_gracefully;  -- we can't do much without the do service
  end if;
  return fd;
end proc;

proc open_notice_server();
  fd := obtain_service ('notice');
  if fd = om then
    help ('Mid-level notice service down - sorry.');
    msg ('could not open "notice" service');
    -- Take this to be a non-fatal error.
  end if;
  return fd;
end proc;

proc quit_gracefully;
  -- Degenerate, since we currently have no pump- or pipe-attached child
  exit_gracefully ([ ]);
end proc;

#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-decode.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.34 vc-push.setl

Service provided:
  push
Client of service:
  image (vc-image.setl, Section A.20)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)
  webutil.setl (Section A.44)

Source code:

```
const yhwh = 'vc-push.setl';

-- Send an infinite multi-part MIME document consisting of a stream of
-- images for the benefit of browsers that support the "server-push"
-- model of yore.  (This has long been in Netscape, but never made it
-- into the Evil Empire Explorer.)

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const server_fd = fileno provide_service ('push');

var clients := {};

loop

  [ready] := select ([{sigterm_fd, server_fd} + domain clients]);

  if sigterm_fd in ready then
    msg (yhwh + ' (' + str pid + ') caught SIGTERM');
    quit_gracefully;
  end if;
```

```
for client = clients(pump_fd) | pump_fd in ready loop
  done_client (pump_fd);
end loop;

if server_fd in ready then
  fd := accept (server_fd);
  if fd ≠ om then
    name := getname fd;
    msg (name+‘ accepted’);
    pump_fd := pump();
    if pump_fd = −1 then
      -- child
      [uri, protocol, mime_headers] := get_request (fd);
      mu := massage_uri uri ? {};
      mu.cmd ?:= ‘JPEG’;  -- someday we may dispatch on content type
      image_fd := obtain_service (‘image’);
      if image_fd = om then
        printa (fd, ‘HTTP/1.0 500 Sorry\r’);
        printa (fd, ‘Server: WEBeye\r’);
        printa (fd, ‘Content-type: text/plain\r’);
        printa (fd, ‘\r’);
        printa (fd, ‘JPEG source temporarily unavailable.\r’);
        printa (fd, ‘Please try later.\r’);
        stop;
      end if;
      printa (fd, ‘HTTP/1.0 200 OK’);
      printa (fd, ‘Server: WEBeye’);
      printa (fd, ‘Content-type: multipart/x-mixed-replace;boundary=EOM’);
      printa (fd, ‘’);
      printa (fd, ‘--EOM’);
      loop doing
        printa (image_fd, ‘JPEG’);
        reada (image_fd, n);
      while not eof do
        image := getn (image_fd, n);
        assert #image = n;
        printa (fd, ‘Content-type: image/jpeg’);
        printa (fd, ‘Content-length: ’+str n+‘’);
```

```
          printa (fd, '');
          printa (fd, image+'');
          printa (fd, '--EOM');
          flush (fd);
          if is_integer mu.rate then
            select (om, mu.rate);  -- delay for mu.rate ms
          end if;
        end loop;
        stop;
      end if;
      -- parent continues here
      close (fd);  -- the child hangs onto this
      client := {};
      client.pump_fd := pump_fd;
      client.name := name;
      clients(pump_fd) := client;
    end if;
  end if;

end loop;

proc done_client (pump_fd);
  msg (clients(pump_fd).name + ' done');
  close (pump_fd);
  clients(pump_fd) := om;
end proc done_client;

proc quit_gracefully;
  exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                              client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
#include "webutil.setl"
```

A.35 vc-quit.setl

Called by parent program:
  vc-restart.setl (Section A.37)
Textually #includes:
  vc-admin.setl (Section A.1)

Source code:

```
const yhwh = 'vc-quit.setl';

-- Stop the Box

const my_lock = 'vc-quitting';  -- lock file (mutex)
const pid_dir = 'vc-pid';
const main_pid_file = pid_dir + '/vc-toplev';
const ps_cmd = 'ps alxwwj';  --- use 'ps -edalfj' on Solaris

commence;  -- acquire mutex or exit abnormally right away

if fexists main_pid_file then
  relieve (main_pid_file, true);
else
  msg ('Pid record "'+main_pid_file+'" not found.');
  -- But that's no reason to abandon the cleanup effort early...
end if;

-- If relieve executed, everything should now be cleaned out, but
-- just to make extra sure...
if fexists pid_dir then
  for pid_file in split (filter ('echo '+pid_dir+'/*')) |
      pid_file notin {'', pid_dir+'/*'} loop
    relieve (pid_file, false);
  end loop;
else
  msg ('Directory "'+pid_dir+'" doesn''t exist???');
  msg ('Something is seriously broken in this installation!');
  msg ('Abending.');
  finis (1);
```

```
end if;

msg ('Calling '+str ps_cmd+' ...');
system (ps_cmd);
msg ('Done.');

finis (0);  -- release mutex and exit normally

proc relieve (pid_file, complain_if_not_found);
 id := val (getfile pid_file ? '0');
 if not is_integer id or id ≤ 0 or id ≥ 2**31 then
   msg ('File '+str pid_file+' does not contain a valid process id!');
 elseif pexists id then
   msg ('Sending TERM signal to pid '+str id+' ...');
   kill (id);  -- the process is supposed to propagate this signal
   -- Wait up to 10 seconds for the process to go away
   loop for i in {1 .. 100} while pexists id do
     select (om, 100);
   end loop;
   -- Clear out anything in the process's process group, in case it
   -- or any of its components orphaned any children
   msg ('Sending TERM to process group '+str id+' ...');
   kill (-id);
   -- Give that polite signal 1414 ms to take effect
   select (om, 1414);
   -- Blast away any misguided limpets
   msg ('Sending KILL to process group '+str id+' ...');
   kill (-id, 'KILL');
 elseif complain_if_not_found then
   msg ('Process id '+str id+' not found.');
 end if;
end proc relieve;

#include "vc-admin.setl"
```

A.36 vc-recv.setl

Called by parent program:
  vc-seq.setl (Section A.39)
Calls child program:
  vc-input.setl (Section A.22)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-msg.setl (Section A.30)

Source code:

```
const yhwh = ‘vc–recv.setl’;

-- Low-level serial input receiver

-- This program tries to build “frames” (Canon VC-C3 messages) from
-- characters received through vc-input.  Each frame received
-- (or created as a result of error detection) is encapsulated as a
-- SETL string and written to stdout on a new line.

const vc_input_cmd = ‘exec setl vc–input.setl’;  -- needed for xrefs
const vc_input = ‘vc–input’;  -- how we actually invoke vc-input.setl

const sigterm_fd = open (‘SIGTERM’, ‘signal’);  -- catch TERM signals
var in_fd := open_input();

loop do
  c := get_char();
  if c = om then
    -- The input source keeps yielding oms.  Quit.
    msg (‘cannot obtain character from ’+vc_input);
    quit_gracefully;
  end if;
  length := abs c;
  if 3 ≤ length and
        length ≤ 32 then
    s := ‘’;
    for i in {1 .. length} loop
```

```
      c := get_char (800);  -- max 800 ms between chars
      if c = om then
        s := '*';  -- meaning "inter-char timeout"
        goto tell_parent;
      end if;
      s +:= c;
    end loop;
    c frome s;  -- remove last char of s and put in c
    if abs c ≠ −(length +/ [abs a : a in s]) mod 256 then
      s := '+';  -- meaning "checksum error"
      goto tell_parent;
    end if;
  else
    -- A bad length byte means serious trouble.  Drain any input
    -- that arrives soon after it, and report a checksum error.
    msg ('bad length byte (0x'+hex c+') ... draining input');
    loop for i in {1 .. 32} while get_char (50) ≠ om do
      pass;  -- any char within 50 ms of its predecessor is "soon"
    end loop;
    s := '+';  -- meaning "checksum error"
    goto tell_parent;
  end if;
tell_parent:
  write (s);
  flush (stdout);
end loop;

-- Get a character within ms milliseconds, or return om on timeout
proc get_char (ms(*));
  max_retries := 10;
  n_retries := 0;
retry:
  n_retries +:= 1;
  if ms(1) ≠ om then
    [ready] := select ([{in_fd, sigterm_fd}], ms(1));
  else
    [ready] := select ([{in_fd, sigterm_fd}]);
  end if;
  if sigterm_fd in ready then
    msg (yhwh + ' (' + str pid + ') caught SIGTERM');
    quit_gracefully;
  end if;
  if in_fd in ready then
    c := getc (in_fd);
    if c = om and n_retries < max_retries then
      --- Do we really want to do this?
      msg (vc_input+' went down!  Restarting it...');
      close (in_fd);
      select (om, 618);  -- wait 0.618 sec before reopening
      in_fd := open_input();
      msg (vc_input+' reopened, in_fd = '+str in_fd);
      goto retry;
    end if;
    return c;
  else
    return om;
  end if;
end proc;

proc open_input();
  in_fd := open (vc_input, 'pipe-from');
  if in_fd = om then
    msg ('could not open '+vc_input);
    quit_gracefully;
  end if;
  return in_fd;
end proc;

proc quit_gracefully;
  exit_gracefully ([[vc_input, in_fd]]);
end proc;

#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.37 vc-restart.setl

Called by parent program:
  vc-cron.setl (Section A.9)
Calls child programs:
  vc-go.setl (Section A.18)
  vc-quit.setl (Section A.35)
Textually #includes:
  vc-admin.setl (Section A.1)

Source code:

```
const yhwh = 'vc-restart.setl';

-- Restart the Box, meaning stop it if necessary, preserve the old log,
-- and start it again

const my_lock  = 'vc-restarting';  -- lock file (mutex)
const vc_lock  = 'vc-lock';  -- Box's lock file
const vc_log   = 'vc-log';   -- Box's log file
const vc_num   = 'vc-number';  -- contains seq.# of last log saved
const vc_go_cmd   = 'exec setl vc-go.setl';
const vc_quit_cmd = 'exec setl vc-quit.setl';

commence;  -- acquire mutex or exit abnormally right away

msg ('Calling '+str vc_quit_cmd+' ...');
quit_rc := system (vc_quit_cmd);
if quit_rc ≠ 0 then
  msg (str vc_quit_cmd+' terminated abnormally.');
  msg ('Also abending.');
  finis (1);
end if;

if lexists vc_lock then
  msg (str vc_quit_cmd+' completed normally but lock file '+vc_lock+
                                              ' still exists!');
  msg ('Removing '+vc_lock+' by force ...');
  unlink (vc_lock);
```

```
end if;

if fexists vc_log then
  n := (val (getfile vc_num ? '0') ? 0) + 1;
  saved_log := vc_log+'.'+str n;
  msg ('Moving '+vc_log+' to '+saved_log+' ...');
  system ('mv '+vc_log+' '+saved_log);
  putfile (vc_num, str n);
end if;

msg ('Calling '+str vc_go_cmd+' ...');
system (vc_go_cmd);
msg ('Done.');

finis (0);  -- release mutex and exit normally

#include "vc-admin.setl"
```

A.38 vc-send.setl

Called by parent program:
  vc-seq.setl (Section A.39)
Calls child programs:
  vc-autoinit.setl (Section A.3)
  vc-clear.setl (Section A.6)
  vc-comdev.setl (Section A.7)
  vc-init.setl (Section A.21)
Textually #includes:
  vc-msg.setl (Section A.30)

Source code:

```
const yhwh = 'vc-send.setl';

-- Low-level sender

-- This program is normally invoked from vc-send, which is
-- simply a setuid'd wrapper compiled from a C program containing
--
-- main() {
--   execl("$(SETL)", "setl", "vc-send.setl", 0);
-- }
--
-- where $(SETL) has been substituted with the absolute pathname of
-- the 'setl' program (the SETL driver) by the Makefile.

com_dev := command_line(1) ? filter ('exec setl vc-comdev.setl');
com_fd := fileno open (com_dev, 'w');

tie (stdin, stdout);

loop doing
  read (frame);
while not eof do
  if #frame = 1 then
    case frame of
    ('i'): system ('exec setl vc-init.setl');
    ('c'):  system ('exec setl vc-clear.setl');
    ('a'):  system ('exec setl vc-autoinit.setl');
    else msg ('Unrecognized special command ' + pretty frame);
    end case;
    -- Some of the above operations will cause our next putc to the
    -- device (com_fd) to fail horribly if we do not do this:
    close (com_fd);
    select (om, 300);  -- wait 300 ms before reopening the device
    com_fd := fileno open (com_dev, 'w');
  else
    csum := #frame + 1 +/ [abs c : c in frame];
    putc (com_fd, char (#frame + 1) + frame + char (-csum mod 256));
    flush (com_fd);
  end if;
  print;
end loop;

#include "vc-msg.setl"
```

A.39 vc-seq.setl

Client of service:
notify (vc-event.setl, Section A.12)
Called by parent program:
vc-model.setl (Section A.27)
Calls child programs:
vc-recv.setl (Section A.36)
vc-send.setl (Section A.38)
Textually #includes:
vc-exit.setl (Section A.15)
vc-msg.setl (Section A.30)
vc-obtain.setl (Section A.31)

Source code:

```
const yhwh = 'vc-seq.setl';

-- Low-level Canon VC-C3 command sequencer

-- This pump takes commands or sequences thereof which are
-- clocked out on attached tick-based schedules.  It also remains
-- receptive at all times to event notices that are generated
-- by the Canon and mixed in with the command responses, which
-- themselves may be delayed.  Advantage is taken of the fact that
-- with the Canon one doesn't necessarily have to wait for the
-- response to one command before sending out another.  So, for
-- example, we can commence a panning and zooming operation at
-- almost the same time, and it might be on the order of seconds
-- before we receive confirmation of the newly accomplished settings.
-- We send all "asynchronously" received notices to the notify
-- service as events, as well as unexpected responses—see the calls
-- to the local notify routine in this program.

const ack_time_limit = 500;  -- ms

-- Low-level receiver:
const in_fd = fileno open ('exec setl vc-recv.setl', 'pipe-from');
```

```
-- Interface to low-level sender, vc-send.setl:
const vc_send_cmd = 'exec setl vc-send.setl';  -- needed for xrefs
const out_fd = fileno open ('vc-send', 'pump');

const note_fd = fileno obtain_service ('notify');  -- event consumer
const sigterm_fd = open ('SIGTERM', 'signal');

open ('SIGPIPE', 'ignore');  -- but see put_frame

tie (stdin, stdout);

loop

  ready := select_or_exit_on_sigterm ({stdin, in_fd});

  if in_fd in ready then
    -- in this state, treat anything from the controller as a note
    frame := get_frame();
    if frame ≠ om then
      notify (frame);
    end if;
  end if;

  if stdin in ready then
    read (seq);  -- command packet, a map
    if eof then
      quit_gracefully;
    end if;
    time_limit := round (seq.time_limit ? 1000);
    if (cmd := seq.cmd) ≠ om then
      response := do_cmd (cmd, time_limit);
      write (response);
    elseif (cmds := seq.cmds) ≠ om then
      responses := do_cmds (cmds, seq.tick_ms, time_limit);
      write (responses);
    else
      msg ('unrecognized seq form ' + str seq + ' ignored');
    end if;
```

```
 end if;

end loop;

proc get_frame();
 reada (in_fd, frame);
 if eof then
  msg ('EOF from ' + str filename in_fd);
  quit_gracefully;
 elseif frame = '*' then  -- inter-char timeout
  return frame;
 elseif frame = '+' then  -- checksum error
  put_frame (unhex '8810');  -- checksum error nak
  return om;
 elseif #frame < 2 then  -- illegal 1-byte frame
  return frame;
 elseif is_ack_or_nak frame then
  return frame;
 else  -- note or response
  ack := char (16#80 bit_or (abs frame(1) bit_and 16#0f)) + '\x00';
  put_frame (ack);  -- ack it, whichever it is
  if is_note frame then
   notify (frame);
   return om;
  else
   return frame;
  end if;
 end if;
end proc get_frame;

proc do_cmd (cmd, time_limit);
 put_frame (cmd);
 if #cmd = 1 then
  return cmd;
 end if;
 awaiting_ack := true;
 response := om;
 deadline := clock + time_limit + 1;
 time_left := time_limit + 1;
 while time_left > 0 loop
  ready := select_or_exit_on_sigterm ({in_fd}, time_left);
  if in_fd in ready then
   frame := get_frame();
   if frame ≠ om then
    if #frame < 2 then  -- inter-char timeout or bad frame
     return frame;  -- abandon command; “response” is this frame
    elseif is_ack_or_nak frame then
     if awaiting_ack then
      if is_ack frame then
       if response ≠ om then
        return response;
       end if;
       awaiting_ack := false;
      else  -- abandon command; “response” is this nak frame
       return frame;
      end if;
     else  -- ack or nak unexpected
      notify (frame);  -- treat as note but otherwise ignore
     end if;
    else  -- other response
     if not awaiting_ack then  -- already got the ack
      return frame;  -- take this frame to be the response
     else  -- didn’t get ack yet (strange but supported)
      response := frame;  -- to be returned when the ack comes
     end if;
    end if;
   end if;
  end if;
  time_left := deadline − clock;
 end loop;
 return ‘!’;  -- our way of indicating this kind of timeout
end proc do_cmd;

proc do_cmds (cmds, tick_ms, time_limit);
 const n = #cmds;
 responses := [ ];
```

```
timer_fd := open (str tick_ms, ‘real-ms’);
tick := 0;
while #cmds > 0 loop
  [ticknum, cmd] fromb cmds;
  while tick < ticknum loop
    ready := select_or_exit_on_sigterm ({timer_fd, in_fd});
    if timer_fd in ready then
      geta (timer_fd, dummy);
      tick +:= 1;
    end if;
    if in_fd in ready then
      frame := get_frame();
      if frame ≠ om then
        responses with:= frame;
      end if;
    end if;
  end loop;
  put_frame (cmd);
  ack_deadline := clock + ack_time_limit + 1;
  ack_time_left := ack_time_limit + 1;
  while ack_time_left > 0 loop
    -- awaiting ack
    ready := select_or_exit_on_sigterm ({in_fd}, ack_time_left);
    if in_fd in ready then
      frame := get_frame();
      if frame ≠ om then
        if is_ack frame then
          ack_time_left := 0;  -- got ack; quit loop
        else
          responses with:= frame;
          if is_nak frame then
            ack_time_left := 0;  -- give up waiting for ack
          end if;
        end if;
      end if;
    else
      responses with:= ‘@’;  -- for “ack timeout”
      ack_time_left := 0;  -- give up waiting for ack
      end if;
      if ack_time_left > 0 then
        ack_time_left := ack_deadline − clock;
      end if;
    end loop;
  end loop;
  deadline := clock + time_limit + 1;
  time_left := time_limit + 1;
  -- Now enter a final stage of just waiting up to the time limit for
  -- the number of responses to reach the original number of commands (n)
  while #responses < n and time_left > 0 loop
    ready := select_or_exit_on_sigterm ({in_fd}, time_left);
    if in_fd in ready then
      frame := get_frame();
      if frame ≠ om then
        responses with:= frame;
      end if;
    end if;
    time_left := deadline − clock;
  end loop;
  close (timer_fd);
  return responses;
end proc do_cmds;

op is_note (frame);
  return is_string frame
     and #frame ≥ 2
     and ((abs frame(1) bit_and 16#80) = 0)
     and ((abs frame(2) bit_and 16#20) ≠ 0);
end op;

op is_ack_or_nak (frame);
  return is_string frame
     and #frame ≥ 2
     and (abs frame(1) bit_and 16#80) ≠ 0;  -- “frame id” byte
end op;

op is_ack (frame);
```

```
 return is_ack_or_nak frame and abs frame(2) = 0;  -- “cmd id” byte
end op;

op is_nak (frame);
 return is_ack_or_nak frame and abs frame(2) ≠ 0;  -- “cmd id” byte
end op;

proc notify (frame);
 writea (note_fd, frame);
 flush (note_fd);
end proc;

proc select_or_exit_on_sigterm (fds, max_time(*));
 if #max_time = 0 then
  [ready] := select ([{sigterm_fd} + fds]);
 else
  [ready] := select ([{sigterm_fd} + fds], max_time(1));
 end if;
 if sigterm_fd in ready then
  msg (yhwh + ‘ (’ + str pid + ‘) caught SIGTERM’);
  quit_gracefully;
 end if;
 return ready;
end proc;

proc put_frame (frame);
 -- If the output operation provokes a SIGPIPE, we remain blissfully
 -- unaware of it because we have explicitly requested open to
 -- “ignore” that signal.  But we can detect disappearance of the
 -- sender nonetheless by checking for EOF (getline out_fd = om),
 -- because normally the sender acks our requests with at least an
 -- empty line:
 writea (out_fd, frame);
 if getline out_fd = om then
  msg (str filename out_fd + ‘ appears to have crashed’);
  quit_gracefully;
 end if;
end proc;
```

```
proc quit_gracefully;
  exit_gracefully ([[str filename in_fd, in_fd],
                   [str filename out_fd, out_fd]]);
end proc;

#include “vc-obtain.setl”
#include “vc-exit.setl”
#include “vc-msg.setl”
```

A.40 vc-simpler.setl

Textually #included by:
  vc-jumper.setl (Section A.25)
  vc-mover.setl (Section A.29)
  vc-zoomer.setl (Section A.43)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)

Source code:

```
-- This file is meant to be #included by others, after they define:
--
-- yhwh  -  the name of the program that #includes this code
-- service_name  -  e.g., 'mover'
-- n_parms  -  1 or 2
-- build_cmd  -  statements to build cmd to send to the do service

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const server_fd = fileno provide_service (service_name);

var clients := {};

loop

 [ready] := select ([{sigterm_fd, server_fd} + domain clients]);

 if sigterm_fd in ready then
  msg (yhwh + ' (' + str pid + ') caught SIGTERM');
  quit_gracefully;
 end if;

 for client = clients(pump_fd) | pump_fd in ready loop
  done_client (pump_fd);
 end loop;
```

```
  if server_fd in ready then
    fd := accept (server_fd);
    if fd ≠ om then
      name := getname fd;
      msg (name+‘ accepted’);
      pump_fd := pump();
      if pump_fd = −1 then
        -- child
        do_fd := fileno obtain_service (‘do’);
        loop
          if (line := getline fd) ≠ om and
             #(t := split (line)) = n_parms and
             forall parm in t | is_num parm then
            build_cmd
            writea (do_fd, cmd);
            geta (do_fd, response);
            printa (fd, response);  -- normally an empty line
          else
            stop;
          end if;
        end loop;
      end if;
      -- parent continues here
      close (fd);
      client := {};
      client.name := name;
      clients(pump_fd) := client;
    end if;
  end if;

end loop;

op is_num (a);
  return a(‘^[+-]?[0-9]+(\\.[0-9]+)?$’) ≠ om;
end op;

proc done_client (pump_fd);
```

```
  msg (clients(pump_fd).name + ' done');
  close (pump_fd);
  clients(pump_fd) := om;
end proc done_client;

proc quit_gracefully;
  exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                                          client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
```

A.41 vc-snap.setl

Service provided:
  snap
Client of service:
  image (vc-image.setl, Section A.20)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-exit.setl (Section A.15)
  vc-getname.setl (Section A.16)
  vc-msg.setl (Section A.30)
  vc-obtain.setl (Section A.31)
  vc-provide.setl (Section A.32)
  webutil.setl (Section A.44)

Source code:

```
const yhwh = 'vc-snap.setl';

-- This server sends a single HTTP-wrapped JPEG image, then closes.

const sigterm_fd = open ('SIGTERM', 'signal');  -- catch TERM signals
const snap_fd = fileno provide_service ('snap');

var clients := {};

loop

  [ready] := select ([{sigterm_fd, snap_fd} + domain clients]);

  if sigterm_fd in ready then
    msg (yhwh + ' (' + str pid + ') caught SIGTERM');
    quit_gracefully;
  end if;

  for client = clients(pump_fd) | pump_fd in ready loop
    done_client (pump_fd);
  end loop;
```

```
 if snap_fd in ready then
  fd := accept (snap_fd);
  if fd ≠ om then
   name := getname fd;
   pump_fd := pump();
   if pump_fd = −1 then
    -- child
    [uri, protocol, mime_headers] := get_request (fd);
    mu := massage_uri uri ? {};
    -- We ignore all details of the GET or POST request.
    image_fd := fileno obtain_service ('image');
    printa (image_fd, 'JPEG');
    reada (image_fd, n);
    image := getn (image_fd, n);
    assert #image = n;
    printa (fd, 'HTTP/1.0 200 OK');
    printa (fd, 'Server: WEBeye');
    printa (fd, 'Content-type: image/jpeg');
    printa (fd, 'Content-length: '+str n);
    printa (fd, 'Pragma: no-cache');
    printa (fd, 'Expires: 0');
    printa (fd, '');
    putc (fd, image);
    stop;
   end if;
   close (fd);  -- child hangs onto this
   client := {};
   client.name := name;
   clients(pump_fd) := client;
  end if;
 end if;

end loop;

proc done_client (pump_fd);
 close (pump_fd);
 clients(pump_fd) := om;
```

```
end proc done_client;

proc quit_gracefully;
  exit_gracefully ([['pump for client ' + client.name, pump_fd] :
                                         client = clients(pump_fd)]);
end proc;

#include "vc-provide.setl"
#include "vc-obtain.setl"
#include "vc-getname.setl"
#include "vc-exit.setl"
#include "vc-msg.setl"
#include "webutil.setl"
```

A.42 vc-toplev.setl

Services provided:
  lookup
  publish
Called by parent program:
  vc-go.setl (Section A.18)
Calls child programs:
  vc-camera.setl (Section A.4)
  vc-do.setl (Section A.11)
  vc-event.setl (Section A.12)
  vc-evjump.setl (Section A.13)
  vc-evzoom.setl (Section A.14)
  vc-giver.setl (Section A.17)
  vc-httpd.setl (Section A.19)
  vc-image.setl (Section A.20)
  vc-jumper.setl (Section A.25)
  vc-mouse.setl (Section A.28)
  vc-mover.setl (Section A.29)
  vc-push.setl (Section A.34)
  vc-snap.setl (Section A.41)
  vc-zoomer.setl (Section A.43)
Textually #includes:
  vc-admin.setl (Section A.1)
  vc-allowed.setl (Section A.2)
  vc-getname.setl (Section A.16)

Source code:

```
const yhwh = 'vc-toplev.setl';

-- This is the primordial program for the Box known as WEBeye.
-- It starts all the servers in the Box, catches their log output,
-- and, if necessary, shuts them down.
--
-- This version tries to bring server management up to a high
-- standard.  It takes advantage of the consistent use of
-- obtain_service and provide_service, and of the idiom by which
-- parents start pump and pipe co-processes, to figure out,
--  based on the SETL source texts, which servers (together with
--  their substituent process trees) transitively depend on which
--  services, and thereby what order to start the servers in.
--
--  Note that a server can provide multiple services, a client can
--  obtain multiple services, and a child program can be instantiated
--  multiply.  Thank goodness for SETL maps.
--
--  The file named in my_lock serves as a lock to make sure we have
--  only one instance of the Box running at one time on the local host.
--
--  The file named in vc_link is for the use of a Web server, and
--  points to (1) a static document saying we are in the process of
--  coming up, (2) a pseudo-document created dynamically after we have
--  fully come up and know our port number, (3) a static document saying
--  we are shutting down, or (4) a static document saying we are down.
--
--  So the external party, probably a CGI script, should simply try
--  to read the link file, and either use what it gets or report that
--  the Box has never been started.  See for example 'vc-master.cgi',
--  from which 'vc.cgi' is instantiated.

const stub = 'vc-toplev';   -- our "base name" for logging purposes

const my_lock       = 'vc-lock';       -- lock file (mutex)
const vc_link       = 'vc-link.html';  -- link to one of these 4:
const starting_name = 'vc-starting.html';
const running_name  = 'vc-running.html';
const stopping_name = 'vc-stopping.html';
const down_name     = 'vc-down.html';
const master_name   = 'vc-up.html';  -- template for running_name file

const pid_dir = 'vc-pid';  -- directory for recording server process ids

var pub_fd := om;      -- miscellaneous file descriptors
var lookup_fd := om;   --  ...
var health_fd := om;   --  ...
var waiter_fd := om;   -- miscellaneous pseudo-fds
var sigterm_fd := om;  -- ...
var wait_time;
var service_db := {};  -- service name ↦ [host,port,pid]
var fd_map := {};      -- fd ↦ server name
var src_names;
var server_map, client_map;
var started := [ ];     -- which servers have been started

commence;  -- acquire mutex or exit abnormally right away

spew (stub+' <starting>');

-- Point the link at the "just in the process of coming up" document:
redirect_link (starting_name);

setpgrp();  -- be a process group leader (see terminate)

-- An external record of our pid for the likes of vc-quit and vc-check:
putfile (pid_dir+'/'+stub, str pid);

sigterm_fd := open ('SIGTERM', 'signal');  -- catch TERM signals

--- This global dependency analysis phase is slow enough that it
--- should probably be moved to "configuration" time as something
--- to be re-done if any source file changes:

-- Raw names of the program sources:
src_names := {src_name in split (filter (
                      "grep -l 'const yhwh' vc-*.setl", ""))
                         | src_name notin {'', yhwh}};

server_map := scan ('provide_service');  -- service name ↦ server name

client_map := scan ('obtain_service');   -- service name ↦ client name

-- The use of 'exec' in front of our invocations of the SETL driver
-- is idiomatic—the shell (/bin/sh) is implicitly used to launch all
-- commands started by open, filter, and system, and sometimes
-- (depending on the shell implementation, but almost always if the
-- command has tricky things like I/O redirections in it, and always
-- if it consists of multiple process specifications) hangs around.
-- This interferes with our desire to send signals such as SIGTERM to
-- our SETL subprocesses, because the shell will not propagate these
-- without being told to.  The easier solution is simply to have the
-- shell move out of the way as soon as it has parsed the command and
-- set up the I/O redirections, and is ready to launch the SETL
-- subprocess (see proc start in this very program for an example):
parent_child_map := {[parent, child] : src_name in src_names,
   line in split (filter (prep_cmd (src_name)+" | "+
    "egrep '(const|open|filter|system).*exec setl .*\\.setl'", ""),
                                               '\n') | #line > 0
  doing
    src_name('\\.setl$') := '';
    parent := src_name;
    line(1 .. 'setl ') := '';
    line('\\.setl' .. ) := '';
    child := line;
};


-- Start core services and identify them with environment variables:
pub_fd    := core_service ('publish', 'VC_PUBLISH');  -- publication
lookup_fd := core_service ('lookup',  'VC_LOOKUP');   -- information

-- Start a warning timer to report services that fail to come up:
waiter_fd := open ('60000', 'real-ms');
wait_time := 0;  -- minutes

-- Melt down server_map and client_map by removing entries from
-- them as services come up.  Also record which servers have been
-- started, in order, so we can later shut them down in reverse order:

msg ('starting servers...');

while #server_map > 0 loop
```

```
  -- Which servers in server_map can now be started?  The
  -- prerequisite is that among the constituent programs of a server
  -- (the server name's transitive closure under parent_child_map),
  -- there are none still in client_map, meaning no clients dependent
  -- on services that are not yet up.

  servers := {server in range server_map | server notin started and
       forall pgm in transitive_closure (parent_child_map, server)
            | pgm notin range client_map};
  start (servers);

  -- Do like the main loop until a new service publishes itself:
  old_service_db := service_db;
  while service_db = old_service_db loop
    main_loop_step;
  end loop;
  service_names := domain (service_db − old_service_db);
  assert #service_names = 1;  -- presume 1 at a time from main_loop_step
  service_name := arb service_names;

  msg ('service "'+service_name+'" is up');

  -- Revise the maps, preparatory to re-evaluating the dependencies:
  server_map(service_name) := om;
  client_map{service_name} := {};

end loop;

close (waiter_fd);  -- finished with the egg timer
waiter_fd := om;

close (pub_fd);  -- unless you'd like to allow further publication
pub_fd := om;

health_fd := core_service ('health', 'VC_HEALTH');   -- sanity check

-- Instantiate the pseudo-document to be presented while we are running:
master := getfile master_name;
```

```
gsub (master, 'LOOKUP', getenv 'VC_LOOKUP');  -- lookup service locus
putfile (running_name, master);  -- master as after instantiation
if getfile running_name ≠ master then
  msg ('fatal – problem creating file "'+running_name+'"');
  terminate (1);
end if;
msg ('created "'+running_name+'"');
-- Point the link at the "now running" pseudo-document.  It's not a
-- "real" document, because all it actually does is give the location
-- of our lookup service for a CGI script to pick up.
redirect_link (running_name);

spew (stub+' <ready>');

loop  -- until terminate is called
  main_loop_step;
end loop;


-- Try to make link file point appropriately for our life-cycle phase
proc redirect_link (target);
  unlink (vc_link);
  clear_error;
  link (target, vc_link);
  if last_error = no_error then
    msg (str vc_link+' now refers to '+str target);
  else
    msg ('problem pointing '+str vc_link+' at '+str target+' – '+
                                        last_error);
  end if;
end proc;

-- "Pre-process" some SETL source
proc prep_cmd (src_name);
  -- First check for an early TERMinate request
  [ready] := select ([{sigterm_fd}], 0);
  if sigterm_fd in ready then
    msg (yhwh + ' (' + str pid + ') caught SIGTERM');
    terminate (0);
  end if;
  return "setl -c "+src_name+" | " +
        "awk '/^%SOURCE/,/^%CODE/' | " +
        "sed -e 's/--.*$//'";
end proc;

-- Obtain a service name ↦ program name (sans '.setl' suffix) map
proc scan (what);
  return {[service_name, src_name] : src_name in src_names,
           line in split (filter (prep_cmd (src_name) +
            " | grep "+what+" | grep -v proc", ""), '\n') | #line > 0
          doing
            service_name := unstr line("'" .. "'");
            src_name('.setl$') := '';
         };
end proc;

-- Start a core service and make its location visible to child
-- processes through an environment variable
proc core_service (serv_name, envt_var);
  var serv_fd, serv_host, serv_port, serv_pid, serv_loc;  -- locals
  serv_fd := open ('0', 'server-socket');  -- listen on arbitrary port
  serv_host := 'localhost';
  serv_port := port serv_fd;
  serv_pid := pid;
  -- Include also a service_db entry for this core service:
  service_db(serv_name) := [serv_host, serv_port, serv_pid];
  serv_loc := serv_host + ':' + str serv_port;
  -- Make the core service visible:
  setenv (envt_var, serv_loc);
  -- This record can be used by parties external to the Box:
  putfile ('vc-tcp/'+serv_name, serv_loc);
  return serv_fd;
end proc;


proc start (servers);
```

```
 for server in servers loop
  -- Set up the command so that the shell will redirect the server's
  -- stderr into its stdout stream.  Then when we use 'pipe-in'
  -- mode to start the server, we'll be able to pick up all the
  -- debugging and diagnostic output that it and its children spew
  -- on stderr, even though any such child may have stdout
  -- redirected for communication with its parent:
  cmd := 'exec setl '+server+'.setl 2>&1';
  fd := open (cmd, 'pipe-in');
  if fd ≠ om then
   fd_map(fd) := server;
   spew (server+' <started>');
   putfile (pid_dir+'/'+server, str pid(fd));  -- record process id
   started(1..0) := [server];  -- insert server at front of list
  else
   msg ('fatal - cannot open pump "'+cmd+'"');
   terminate (1);
   stop 1;  -- in case terminate mistakenly returns
  end if;
 end loop;
end proc;


proc main_loop_step;

 [ready] := select ([{pub_fd, lookup_fd, health_fd,
                      sigterm_fd, waiter_fd} + domain fd_map]);

 if pub_fd ≠ om and pub_fd in ready then
  fd := accept (pub_fd);
  if fd ≠ om then
   if allowed (fd) then
    reada (fd, service_name, service_info);
    service_db(service_name) := service_info;
   else
    msg (getname fd+' is trying to provide a service???');
   end if;
   close (fd);
  end if;
end if;

if lookup_fd in ready then
  fd := accept (lookup_fd);
  if fd ≠ om then
    if allowed (fd) then
      -- Local clients are expected to make at most a few rapid
      -- lookup requests and then immediately close the connection.
      loop doing
        reada (fd, service_name);
      while not eof do
        printa (fd, service_db(service_name) ? [ ]);
      end loop;
    else
      msg ('refusing lookup service to '+getname fd);
    end if;
    close (fd);
  end if;
end if;

if health_fd in ready then
  fd := accept (health_fd);
  if fd ≠ om then
    if allowed (fd) then
      -- Placeholder for any global checks we want this program to do
      printa (fd, 'ok');  -- faith in self-health
    else
      msg ('refusing health-check service to '+getname fd);
    end if;
    close (fd);
  end if;
end if;

if waiter_fd ≠ om and waiter_fd in ready then
  reada (waiter_fd);
  wait_time +:= 1;
  msg ('services '+str domain server_map+' not started after '+
      str wait_time+‘ minute(s) – still waiting’);
  end if;

  if sigterm_fd in ready then
    msg (yhwh + ‘(’ + str pid + ‘) caught SIGTERM’);
    terminate (0);
  end if;

  for fd in ready | (server := fd_map(fd)) ≠ om loop
    if (s := getline fd) ≠ om then
      spew (server+‘ : ’+s);
    else
      msg (server+‘ exited! – shutting down Box...’);
      terminate (1);
      -- The following code is not executed (terminate does not
      -- return), but this is what should happen if there comes to be
      -- some valid reason for individual servers to terminate:
      close (fd);
      spew (server+‘ <done>’);
      fd_map(fd) := om;
    end if;
  end loop;

end proc main_loop_step;


proc terminate (rc);

  spew (stub+‘ <stopping>’);

  -- Point at the “just in the process of shutting down” document:
  redirect_link (stopping_name);
  -- Get rid of the dynamically created pseudo-document.  If other
  -- processes happen to be reading it, it won’t actually disappear
  -- until they all close it:
  system (‘rm -f ’+running_name);
  msg (‘removed "’+running_name+‘"’);
```

```
if health_fd ≠ om then
  close (health_fd);
  health_fd := om;
end if;

if lookup_fd ≠ om then
  close (lookup_fd);
  lookup_fd := om;
end if;

if pub_fd ≠ om then
  close (pub_fd);
  pub_fd := om;
end if;

inv_fd_map := {[server, fd] : server = fd_map(fd)};

-- Try the polite signal first, to give servers a chance to clean up:
for server in started loop
  fd := inv_fd_map(server);
  msg ('sending TERM signal to '+server+' (pid '+str pid (fd)+')');
  kill (pid (fd));
end loop;

-- Wait for all the servers to go down.  Assume progress is being
-- made as long as no more than 1.618 seconds of silence goes by:
while #fd_map > 0 loop

  [ready] := select ([domain fd_map], 1618);

  if #ready = 0 then
    -- Timeout.  Resort to the impolite signal for remaining servers:
    msg (str range fd_map+' did not exit – killing...');
    for server = fd_map(fd) loop
      kill (pid(fd), 'KILL');
      close (fd);
      spew (server+' <killed>');
      fd_map(fd) := om;
    end loop;

  else
    -- Response from server.  It might be telling us something we
    -- should log before it goes down, or EOF to say it has exited:
    for fd in ready loop
      server := fd_map(fd);
      if (s := getline fd) ≠ om then
        spew (server+‘ : ’+s);
      else
        close (fd);
        spew (server+‘ <done>’);
        fd_map(fd) := om;
      end if;
    end loop;
  end if;

end loop;

-- Lest some servers abandoned their children, make sure all the
-- processes in the Box receive a TERM signal and then a KILL.
-- This is predicated on the assumption that all the processes in
-- the Box are in our process group.  To avoid killing ourself,
-- we do the signalling from a special child that puts itself in
-- its own process group:
if fork() = 0 then
  -- Special child process
  box_pgrp := getpgrp();
  setpgrp();  -- escape the Box's process group
  kill (−box_pgrp);  -- send TERM to all processes in the Box's group
  select (om, 618);  -- wait 0.618 sec (should be plenty)
  kill (−box_pgrp, ‘KILL’);  -- kill them if nothing else did it
  -- Point at the “now down” document:
  redirect_link (down_name);
  spew (stub+‘ <done>’);
  finis (rc);
end if;
```

```
 -- It is not an error for the parent to reach here, nor is it an
 -- error for it not to.  It just depends on whether the special
 -- child above gets to us first or not—a race where we don’t
 -- care who wins...
--- I’m not really quite comfortable with that.  It might be better
--- to spawn the child which starts its own process group right near
--- the beginning, and have it start the tree.  Then the parent should
--- wait for that child to exit, and finally do the group-signalling
--- and THEN the lock release.
 stop rc;

end proc terminate;


proc transitive_closure (f, x);
 -- adapted from SDDS 1986, page 334
 to_process := seen_already := {x};
 return {y : doing
              y from to_process;
              to_process +:= f{y} − seen_already;
              seen_already +:= f{y};
            until #to_process = 0};
end proc;


#include “vc-getname.setl”
#include “vc-allowed.setl”
#include “vc-admin.setl”
```

A.43 vc-zoomer.setl

Service provided:
  zoomer
Client of service:
  do (vc-do.setl, Section A.11)
Called by parent program:
  vc-toplev.setl (Section A.42)
Textually #includes:
  vc-simpler.setl (Section A.40)

Source code:

```
const yhwh = 'vc–zoomer.setl';

-- Simplified "Zoom To" command interface.

-- The client just sends a number (the zoom factor) on each line.

-- The name of the service we provide:
#define service_name 'zoomer'

-- The number of parameters on the command:
#define n_parms  1

-- The full details of the command we will send to the do service:
#define build_cmd \
  cmd := {}; \
  cmd.name := 'Zoom'; \
  cmd.subcmd := 'To'; \
  cmd.zoom_factor := val t(1);

#include "vc-simpler.setl"
```

A.44 webutil.setl

Textually #included by:

vc-httpd.setl (Section A.19)
vc-push.setl (Section A.34)
vc-snap.setl (Section A.41)

Source code:

```
-- Read and parse a GET or POST command into a triple consisting of
-- a URI string, a protocol string (such as 'HTTP/1.0' or 'HTTP/1.1'),
-- and a MIME-header map:
--
proc get_request (fd);
 s := getline fd ? '';
 s('^\r') := ''; s('\r$') := '';  -- kill any \n\r or \r\n effects
 if #(t := split (s)) = 0 then return [ ]; end if;
 [cmd, uri, protocol] := t;
 if to_upper cmd notin {'GET', 'POST'} then return [ ]; end if;
 uri ?:= '/';
 protocol ?:= '';
 mime_headers := {};
 if (to_upper protocol)('^HTTP') ≠ om then
  -- Read more lines
  k := 0;
  loop doing
   s := getline fd ? '';
   s('^\r') := ''; s('\r$') := '';  -- kill \n\r or \r\n effects
  while #split (s) > 0 do
   h := break (s, ':');
   if s('^:') ≠ om then
    s('^:[ \t]*') := '';
    mime_headers with:= [h, s];
    if to_lower h = 'content-length' and s('^[0-9]') ≠ om then
     k := val s('^[0-9]*');
    end if;
   end if;
  end loop;
  if k > 0 then
```

```
      content := getn (fd, k) ? ‘’;
      if #content ≠ k then
        printa (stderr, myname, ‘:’,
               ‘expected’, k, ‘bytes of content but got’, #content);
      end if;
      uri +:= ‘?’ + content;
    end if;
  end if;
  return [uri, protocol, mime_headers];
end proc get_request;

-- Parse a URI into a map, such that ‘/’, ‘?’, and ‘&’ are taken as
-- delimiters between map elements, and ‘.’ and ‘=’ are taken to
-- separate a key and associate within each map element.
--
-- The first delimited element is treated specially:  its key is
-- taken to be ‘cmd’, and the whole element is the associate.
--
-- Also, any element consisting of a comma-separated pair of
-- unsigned integers is taken to be the associate of key ‘click’.
--
-- Otherwise, a missing associate is taken to be the null string.
--
-- In elements containing multiple separators, only the first is
-- recognized, and what remains is taken to be the associate.
--
op massage_uri (uri);
  if uri = om or uri = ‘’ or uri = ‘/’ then
    return {};  -- null map, no command or parameters
  end if;
  if not is_string uri or uri(1) ≠ ‘/’ then
    return om;  -- I simply cannot respect this as a URI
  end if;
  r := {};
  uri(1) := ‘’;
  r.cmd := sub (uri, ‘^[^/?&]*’);
  r +:= {general_pair unescape p : p in split (uri, ‘[/?&]’)};
  return r;
```

```
end op;

op general_pair (p);
  if #p = 0 then return om; end if;
  if p('^[0-9]+,[0-9]+$') ≠ om then
    return ['click', numericize p];
  end if;
  x := sub (p, '^[^=.]*');
  y := if #p > 0 then p(2..) else '' end;
  return [x, numericize y];
end op;

op numericize (s);
  if s('^[-+]?[0-9]+$') ≠ om then
    return val s;
  elseif s('^[-+]?[0-9]*[.][0-9]*$') ≠ om then
    return val s;
  elseif ',' in s then
    return [numericize x : x in split(s,',')];
  else
    return s;
  end if;
end op;

op escape (x);  -- convert certain characters to %HH (hex)
  return '' +/ [if c in '~`!@#$%^&*()+=[]{};:\'"<>?\\|' then '%' + hex c
              else c end : c in x];
end op;

op escape_map (f);  -- escape a SETL map for Web transmission
  r := '' +/ ['&' + x + '=' + escape y : y = f(x)];
  if #r > 0 then r(1) := ''; end if;
  gsub (r, ' ', '+');
  return r;
end op;

op unescape (x);  -- convert %HH hex escapes to normal chars
  r := '';
  while #x > 2 loop
    if x(1) = ‘%’ and x(2 .. 3)(‘[0-9a-fA-F][0-9a-fA-F]’) ≠ om then
      r +:= unhex x(2 .. 3);
      x(1 .. 3) := ‘’;
    else
      r +:= len (x, 1);
    end if;
  end loop;
  r +:= x;
  return r;
end op;

op unescape_map (s);  -- convert Web form-type spam to a SETL map
  gsub (s, ‘\\+’, ‘ ’);  -- change plusses to blanks
  r := {pair unescape x : x in split (s, ‘\\&’)};  -- decode
end op;

op pair (x);
  s := break (x, ‘=’);  -- s := part of x before ‘=’ or all of x; x := x(‘=’ .. ) or ‘’
  return [s, if #x > 0 then x(2 .. ) else ‘’ end];
end op;
```

Bibliography

[1] Ada Core Technologies, Inc., 1998. At http://www.gnat.com/. Home page for the Ada 95 compilation system, GNAT.

[2] NYU Ada Project. Ada/Ed interpreter: Executable semantic model for Ada. Technical report, Courant Institute of Mathematical Sciences, New York University, July 1984. Self-documenting listing of the NYU Ada/Ed Compiler, Version 1.4, validated 28 June 1984.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] D. Aliffi, D. Montanari, and E.G. Omodeo. Meta-interpreting SETL. In *SED: A SETL-Based Prototyping Environment* [125], October 1988.

[5] F.E. Allen. Program optimization. In Mark I. Halpern and Christopher J. Shaw, editors, *Annual Review in Automatic Programming*, volume 5, pages 239–307. Pergamon Press, New York, 1969.

[6] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[7] Frances E. Allen. A basis for program optimization. In *Proc. IFIP Congress 71*, pages 385–390. North-Holland, 1972.

[8] Frances E. Allen. Interprocedural data flow analysis. In *Proc. IFIP Congress 74*, pages 398–402. North-Holland, 1974.

[9] Frances E. Allen. A method for determining program data relationships. In Andrei Ershov and Valery A. Nepomniaschy, editors, *Proc. International Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972*, volume 5 of *Lecture Notes in Computer Science*, pages 299–308. Springer-Verlag, 1974.

[10] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

[11] Frances E. Allen and John Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1972.

[12] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[13] David Bacon. Dave's famous original SETL server, 1994. At http://birch.eecs.lehigh.edu/~bacon/setl-server.html. Allows security-restricted SETL programs to be edited or fetched and then run on a server host, from within a browser environment.

[14] David Bacon. Dewar Online!, 1994. At http://birch.eecs.lehigh.edu/cgi-bin/html?dewar-online.html. A whimsical interface to the comp.lang.ada newsgroup.

[15] David Bacon. The SETL home page, 1994. At http://birch.eecs.lehigh.edu/~bacon/. My home page has long claimed to be the home of SETL, and touted SETL as the "world's most wonderful programming language".

[16] David Bacon. LabEye, 1997. At http://birch.eecs.lehigh.edu:6565/. View of an oscilloscope and a pair of bicolored LEDs that can be controlled through a browser.

[17] David Bacon. LogEye, 1997. At http://birch.eecs.lehigh.edu:8009/imp. Log-map image of the view through a videocamera mounted on a spherical pointing motor [26, 25].

[18] David Bacon. MUReye—a movable, zoomable web camera, 1999. At http://128.180.98.223/cgi-bin/MUReye/. An instantiation of WEBeye (see Chapter 4 of this dissertation).

[19] David Bacon. SETL library documentation, 1999. At http://birch.eecs.lehigh.edu/~bacon/setl-doc.html.

[20] David Bacon. Slim, 1999. At http://birch.eecs.lehigh.edu/slim/. Starting point for my adaptation of the documentation and distribution files comprising Herman Venter's Slim [204] language system, which he no longer maintains.

[21] Bernard Banner. Private communication, 1999.

[22] Nancy Baxter, Ed Dubinsky, and Gary Levin. *Learning Discrete Mathematics with ISETL*. Springer-Verlag, 1989.

[23] Nancy Hood Baxter. Understanding how students acquire concepts underlying sets. In James J. Kaput and Ed Dubinsky, editors, *Research Issues in Undergraduate Mathematics Learning*, number 33 in MAA Notes, pages 99–106. The Mathematical Association of America, Washington, DC, 1994.

[24] David M. Beazley. SWIG (simplified wrapper and interface generator), 1999. At http://www.swig.org/.

[25] B.B. Bederson, R.S. Wallace, and E.L. Schwartz. A miniature pan-tilt actuator: The spherical pointing motor. *IEEE Journal of Robotics and Automation*, 10(3):298–308, June 1994. Also published by the Courant Institute of Mathematical Sciences at New York University as Computer Science Technical Report No. 601-R264, April 1992.

[26] Benjamin B. Bederson, Richard S. Wallace, and Eric L. Schwartz. Spherical pointing motor. United States Patent No. 5,204,573, April 1993.

[27] V.H. Bistiolas, C.T. Davarakis, and A. Tsakalidis. Using SETL language for cartography applications based on computational geometry algorithms. In *SED: A SETL-Based Prototyping Environment* [125], 1989.

[28] Bard Bloom and Robert Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[29] C. Bouzas, J. Gazofalakis, P. Spizakis, V. Tampakas, and V. Tziantafillou. SETL-MON: The SETL monitor and performance evaluator. In *SED: A SETL-Based Prototyping Environment* [125], 1989.

[30] J. Cai, Ph. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Moeller, editor, *Constructing Programs from Specifications*, pages 126–164. North-Holland, 1991.

[31] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, November 1992.

[32] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1988/89.

[33] Jiazhen Cai and Robert Paige. Towards increased productivity of algorithm implementation. In David Notkin, editor, *Proc. First ACM SIGSOFT Symposium on the Foundations of Software Engineering* (SIGSOFT '93), volume 18, number 5 of *Software Engineering Notes*, pages 71–78. Association for Computing Machinery Special Interest Group on Software Engineering, December 1993.

[34] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1–2):189–228, July 1995.

[35] Canon USA, Inc. VC-C3 communication camera, 1998. At http://www.usa.canon.com/corpoffice/viscommeq/vcc3.html.

[36] Domenico Cantone and Alfredo Ferro. Techniques of computable set theory with applications to proof verification. *Communications on Pure and Applied Mathematics*, 48(9–10):901–945, September 1995.

[37] Domenico Cantone, Alfredo Ferro, and Eugenio Omodeo. *Computable Set Theory*. Clarendon Press, Oxford, 1989.

[38] Chia-Hsiang Chang and Robert Paige. From regular expressions to DFA's using compressed NFA's. *Theoretical Computer Science*, 178(1–2):1–36, May 1997.

[39] Nigel Paul Chapman. *Theory and Practice in the Construction of Efficient Interpreters*. PhD thesis, University of Leeds, 1980.

[40] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, fourth edition, 1997.

[41] John Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, July 1970.

[42] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[43] John Cocke and Raymond E. Miller. Some analysis techniques for optimizing computer programs. In *Proc. 2nd Hawaii International Conference on System Sciences*, pages 143–146, Honolulu, HI, January 1969.

[44] John Cocke and Jacob T. Schwartz. *Programming Languages and their Compilers*. Courant Institute of Mathematical Sciences, New York University, April 1970.

[45] Daniel E. Cooke. An introduction to SequenceL: A language to experiment with constructs for processing nonscalars. *Software Practice and Experience*, 26(11):1205–1246, November 1996.

[46] Daniel E. Cooke. SequenceL provides a different way to view programming. *Computer Languages*, 24(1):1–32, 1998.

[47] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[48] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[49] Mike Cowlishaw. The Rexx language, 1999. At http://www2.hursley.ibm.com/Rexx/.

[50] M. Davis and J. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. *Comp. Math. Appl.*, 5:217–230, 1979.

[51] Thierry Despeyroux. Executable specification of static semantics. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types: Proc. International Symposium, Sophia-Antipolis, France, June 1984*, volume 173 of *Lecture Notes in Computer Science*, pages 215–233. Springer-Verlag, 1984.

[52] Thierry Despeyroux. TYPOL: A formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[53] R.B.K. Dewar. *The SETL Programming Language*. Courant Institute of Mathematical Sciences, New York University, 1979. Also at http://birch.eecs.lehigh.edu/~bacon/setlprog.ps.gz.

[54] R.B.K. Dewar and A.P. McCann. MACRO SPITBOL—a SNOBOL4 compiler. *Software Practice and Experience*, 7:95–113, 1977.

[55] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[56] Robert B.K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems*, 1(1):27–49, July 1979.

[57] Edsger W. Dijkstra. Notes on structured programming. In *Structured Programming*, pages 1–82. Academic Press, 1972.

[58] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[59] E.-E. Doberkat, E. Dubinsky, and J.T. Schwartz. Reusability of design for complex programs: An experiment with the SETL optimizer. In *Proc. ITT Workshop on Reusability of Software*, pages 106–108, Providence, RI, 1983. ITT.

[60] E.-E. Doberkat and U. Gutenbeil. Prototyping and reusing software. In L. Dusink and P. Hall, editors, *Software Re-Use, Utrecht 1989*, pages 77–86. Springer-Verlag, 1991.

[61] E.-E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E—a prototyping system based on sets. In W. Zorn, editor, *Proc. TOOL90*, pages 109–118. University of Karlsruhe, November 1990.

[62] E.-E. Doberkat, W. Hasselbring, and C. Pahl. Investigating strategies for cooperative planning of independent agents through prototype evaluation. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models: Proc. First International Conference, COORDINATION '96, Cesena, Italy, April 1996*, volume 1061 of *Lecture Notes in Computer Science*, pages 416–419. Springer-Verlag, 1996. A longer version was published as University of Dortmund Software-Technik Memo Nr. 86, December 1995, available at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Doberkat-Hasselbring-Pahl-Memo-86.ps.gz.

[63] E.-E. Doberkat and Kio C. Hyun. Inline expansion of SETL procedures. *ACM SIGPLAN Notices*, 20(12):77–87, 1985.

[64] E.-E. Doberkat and H.-G. Sobottka. A set-oriented program description language for Ada. In R. Prieto-Diaz, W. Schäfer, J. Cramer, and S. Wolf, editors, *Proc. First International Workshop on Software Reusability*, pages 193–196, July 1991.

[65] Ernst E. Doberkat. Efficient translation of SETL programs. In *Proc. 18th Hawaii International Conference on System Sciences*, volume II, pages 457–465, January 1985.

[66] Ernst-Erich Doberkat. A proposal for integrating persistence into the prototyping language SETL/E. Technischer Bericht (technical report) 02-90, University of Essen Computer Science / Software Engineering, April 1990.

[67] Ernst-Erich Doberkat. Integrating persistence into a set-oriented prototyping language. *Structured Programming*, 13(3):137–153, 1992.

[68] Ernst-Erich Doberkat and Dietmar Fox. *Software-Prototyping mit SETL*. Teubner-Verlag, Stuttgart, 1989.

[69] Ernst-Erich Doberkat, Dietmar Fox, and Ulrich Gutenbeil. Translating SETL into Ada, and creating libraries of data structures. In *SED: A SETL-Based Prototyping Environment* [125], 1989.

[70] Ernst-Erich Doberkat, Wolfgang Franke, Ulrich Gutenbeil, Wilhelm Hasselbring, Ulrich Lammers, and Claus Pahl. ProSet—prototyping with sets: Language definition. Technischer Bericht (technical report) 02-92, University of Essen Computer Science / Software Engineering, April 1992. Also at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Essener-Berichte/02-92.ps.gz.

[71] Ernst-Erich Doberkat, Wolfgang Franke, Ulrich Gutenbeil, Wilhelm Hasselbring, Ulrich Lammers, and Claus Pahl. ProSet—a language for prototyping with sets. In Nick Kanopoulos, editor, *Proc. 3rd International Workshop on Rapid System Prototyping*, pages 235–248. IEEE Computer Society Press, Research Triangle Park, NC, June 1992.

[72] Ernst-Erich Doberkat and Ulrich Gutenbeil. SETL to Ada—tree transformations applied. *Information and Software Technology*, 29(10):548–557, December 1987.

[73] V. Donzeau-Gouge, C. Dubois, P. Facon, and F. Jean. Development of a programming environment for SETL. In H.K. Nichols and D. Simpson, editors, *ESEC '87: Proc. 1st European Software Engineering Conference, Strasbourg, France, September 1987*, volume 289 of *Lecture Notes in Computer Science*, pages 21–32. Springer-Verlag, 1987.

[74] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, 1984.

[75] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[76] E. Dubinsky. ISETL: A programming language for learning mathematics. *Communications on Pure and Applied Mathematics*, 48(9–10):1027–1051, September 1995. Presented at NYU on the occasion of Jack Schwartz's 65th birthday.

[77] Ed Dubinsky, Stefan Freudenberger, Edith Schonberg, and J.T. Schwartz. Reusability of design for large software systems: An experiment with the SETL optimizer. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 275–293. ACM Press, New York, 1989.

[78] Ed Dubinsky and Guershon Harel. The nature of the process conception of function. In Harel and Dubinsky [100], pages 85–106.

[79] Ed Dubinsky and Uri Leron. *Learning Abstract Algebra with ISETL*. Springer-Verlag, 1994.

[80] J. Earley. High level iterators and a method of data structure choice. *Computer Languages*, 1(4):321–342, 1975.

[81] William E. Fenton and Ed Dubinsky. *Introduction to Discrete Mathematics with ISETL*. Springer-Verlag, 1996.

[82] Amelia C. Fong. Generalized common subexpressions in very high level languages. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 48–57, January 1977.

[83] Amelia C. Fong. Inductively computable constructs in very high level languages. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, pages 21–28, January 1979.

[84] Amelia C. Fong, John B. Kam, and Jeffrey D. Ullman. Application of lattice algebra to loop optimization. In *Proc. 2nd ACM Symposium on Principles of Programming Languages*, pages 1–9, January 1975.

[85] Amelia C. Fong and Jeffrey D. Ullman. Induction variables in very high level languages. In *Proc. 3rd ACM Symposium on Principles of Programming Languages*, pages 104–112, January 1976.

[86] Amelia C. Fong and Jeffrey D. Ullman. Finding the depth of a flow graph. *Comput. Syst. Sci.*, 15:300–309, 1977.

[87] Free Software Foundation. GNU's not Unix!—the GNU project and the Free Software Foundation (FSF), 1999. At http://www.gnu.org/.

[88] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.

[89] Matthew Fuchs. Escaping the event loop: An alternative control structure for multi-threaded GUIs. In C. Unger and L.J. Bass, editors, *Engineering for HCI*. Chapman & Hall, 1996. From *Engineering the Human Computer Interface* (EHCI '95), and available via http://www.cs.nyu.edu/phd_students/fuchs/.

[90] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[91] Deepak Goyal. An improved intra-procedural may-alias analysis algorithm. Technical Report 777, Courant Institute of Mathematical Sciences, New York

University, February 1999. Also at http://cs1.cs.nyu.edu/phd_students/deepak/publications/improvement.ps.

[92] Deepak Goyal. *A Language Theoretic Approach To Algorithms*. PhD thesis, New York University, January 2000.

[93] Deepak Goyal and Robert Paige. The formal reconstruction and improvement of the linear time fragment of Willard's relational calculus subset. In R. Bird and L. Meertens, editors, *IFIP TC2 Working Conference 1997*, Algorithmic Languages and Calculi, pages 382–414. Chapman and Hall, 1997.

[94] Deepak Goyal and Robert Paige. A new solution to the hidden copy problem. In Giorgio Levi, editor, *Static Analysis: Proc. 5th International Symposium, SAS '98, Pisa, Italy, September 1998*, volume 1503 of *Lecture Notes in Computer Science*, pages 327–348. Springer-Verlag, 1998.

[95] NYU Griffin Project. The Griffin programming language, 1996. At ftp://cs.nyu.edu/pub/griffin/.

[96] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986.

[97] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[98] Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend. *Graphics Programming in Icon*. Peer-to-Peer Communications, 1998.

[99] R.E. Griswold, J.F. Poage, and I.P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1971.

[100] Guershon Harel and Ed Dubinsky, editors. *The Concept of Function: Aspects of Epistemology and Pedagogy*. Number 25 in MAA Notes. The Mathematical Association of America, Washington, DC, 1992.

[101] M.C. Harrison. BALM-SETL: A simple implementation of SETL. *SETL Newsletters* [186], No. 1, November 1970.

[102] W. Hasselbring. On integrating generative communication into the prototyping language ProSet. Technischer Bericht (technical report) 05-91, University of Essen Computer Science / Software Engineering, December 1991. Also at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Essener-Berichte/05-91.ps.gz.

[103] W. Hasselbring. Translating a subset of SETL/E into SETL2. Technischer Bericht (technical report) 02-91, University of Essen Computer Science / Software Engineering, January 1991.

[104] W. Hasselbring. A formal Z specification of ProSet-Linda. Technischer Bericht (technical report) 04-92, University of Essen Computer Science / Software Engineering, September 1992. Also at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Essener-Berichte/04-92.ps.gz.

[105] W. Hasselbring. Animation of Object-Z specifications with a set-oriented prototyping language. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop: Proc. 8th Z User Meeting, Cambridge, UK*, Workshops in Computing, pages 337–356. Springer-Verlag, June 1994. Also published as University of Dortmund Software Technology UniDO Forschungsbericht (research report) 523/1994, available at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring-UniDo-523-1994.ps.gz.

[106] W. Hasselbring and R.B. Fisher. Investigating parallel interpretation-tree model matching algorithms with ProSet-Linda. Software-Technik Memo 77, University of Dortmund, December 1994. Also at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring-Fisher_SWT-Memo-77.ps.gz.

[107] W. Hasselbring and R.B. Fisher. Using the ProSet-Linda prototyping language for investigating MIMD algorithms for model matching in 3-D computer vision. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly*

Structured Problems: Proc. Second International Workshop, IRREGULAR '95, Lyon, France, September 1995, volume 980 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 1995. Also published as University of Dortmund Software Technology UniDO Forschungsbericht (research report) 579/1995, available at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring-Fisher-Irregular95.ps.gz.

[108] W. Hasselbring, P. Jodeleit, and M. Kirsch. Implementing parallel algorithms based on prototype evaluation and transformation. Software-Technik Memo 93, University of Dortmund, January 1997. Also at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring-Jodeleit-Kirsch_SWT-Memo-93.ps.gz.

[109] Wilhelm Hasselbring. Prototyping parallel algorithms with ProSet-Linda. In Jens Volkert, editor, *Parallel Computation: Proc. Second International ACPC Conference, Gmunden, Austria, October 1993*, volume 734 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 1993. Also published as University of Essen Computer Science / Software Engineering Technischer Bericht (technical report) 04-93, 1993, available at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Essener-Berichte/04-93.ps.gz.

[110] Wilhelm Hasselbring. Approaches to high-level programming and prototyping of concurrent applications. Software-Technik Memo 91, University of Dortmund, January 1997. Also at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring_SWT-Memo-91.ps.gz.

[111] Wilhelm Hasselbring. The ProSet-Linda approach to prototyping parallel systems. *Journal of Systems and Software*, 43(3):187–196, November 1998. Also published as University of Dortmund Software Technology UniDO Forschungsbericht (research report) 650/1997, available at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring-UniDo-650-1997.ps.gz.

[112] Wilhelm Hasselbring and Andreas Kröber. Combining OMT with a prototyping approach. *Journal of Systems and Software*, 43(3):177–185, November 1998. Also published as University of Dortmund Software Technology

UniDO Forschungsbericht (research report) 651/1997, available at ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring-Kroeber-UniDo-651-1997.ps.gz.

[113] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[114] C.A.R. Hoare. Data reliability. In *Proc. of the International Conference on Reliable Software*, pages 528–533, 1975. IEEE Cat. No. 75CH0940-7CSR.

[115] Jim Hugunin. JPython home, 1999. At http://www.jpython.org/.

[116] IANA. Internet Assigned Numbers Authority, 1998. At http://www.iana.org/.

[117] IEEE. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities, Volumes 1–2*. Institute of Electrical and Electronics Engineers, 1993. IEEE/ANSI Std 1003.2-1992 & 1003.2a-1992, or ISO/IEC 9945-2.

[118] IEEE. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*. Institute of Electrical and Electronics Engineers, 1996. IEEE/ANSI Std 1003.1, or ISO/IEC 9945-1.

[119] IEEE. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Protocol Independent Interfaces (PII)*. Institute of Electrical and Electronics Engineers, March 1998. P1003.1g, D6.6 (draft standard).

[120] K.E. Iverson. *A Programming Language*. John Wiley, New York, 1962.

[121] Eric F. Johnson and Kevin Reichard. *Advanced X Window Applications Programming*. M&T Books, New York, second edition, 1994.

[122] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International (UK), 1993.

[123] Marc A. Kaplan and Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. In *Proc. 5th ACM Symposium on Principles of Programming Languages*, pages 60–75, 1978.

[124] J. Keller. *Cantor: A Tutorial and a User's Guide*. Education Report 94/9. Kepler, Paris, 1994.

[125] J.P. Keller. SED: A SETL-based prototyping environment. Final report of the SED project, ESPRIT, February 1989.

[126] J.P. Keller and R. Paige. Program derivation with verified transformations—a case study. *Communications on Pure and Applied Mathematics*, 48(9–10):1053–1113, September 1995.

[127] Yo Keller. *An Introduction to Cantor Version 0.41*. Kepler, Paris, February 1991.

[128] Richard Kelsey, William Clinger, and editors Jonathan Rees. Revised(5) report on the algorithmic language Scheme, 1998. Under http://www-swiss.ai.mit.edu/~jaffer/Scheme.html.

[129] Kempe Software Capital Enterprises. Ada and the Web and Java, 1998. At http://www.adahome.com/Resources/Ada_Java.html.

[130] Ken Kennedy. A global flow analysis algorithm. *International Journal of Computer Mathematics*, 3:5–15, December 1971.

[131] Ken Kennedy. Node listings applied to data flow analysis. In *Proc. 2nd ACM Symposium on Principles of Programming Languages*, pages 10–21, January 1975.

[132] Ken Kennedy. A comparison of two algorithms for global data flow analysis. *SIAM J. Comput.*, 5(1):158–180, March 1976.

[133] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[134] Khoral Research Inc. Khoros, 1999. At http://www.khoral.com/.

[135] Philippe Kruchten, Edmond Schonberg, and Jacob Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, 1(4):66–75, October 1984.

[136] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1997.

[137] C.H. Lindsey and S.G. van der Meulen. *Informal Introduction to Algol 68*. North-Holland, 1977.

[138] Zhiqing Liu. *Lazy SETL Debugging with Persistent Data Structures*. PhD thesis, New York University, November 1994.

[139] Zhiqing Liu. A persistent runtime system using persistent data structures. In *ACM Symposium on Applied Computing*, pages 429–436, February 1996.

[140] Zhiqing Liu. A system for visualizing and animating program runtime histories. In *IEEE Symposium on Visual Languages*, pages 46–53. IEEE Computer Society Press, September 1996.

[141] Zhiqing Liu. An advanced C++ library for symbolic computing. Technical Report TR-CIS-0299-11, Purdue University, Indianapolis, IN, February 1999.

[142] J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999. Under http://cs.tu-berlin.de/journal/jflp/articles/1999/A99-03/A99-03.html.

[143] James Low and Paul Rovner. Techniques for the automatic selection of data structures. In *Proc. 3rd ACM Symposium on Principles of Programming Languages*, pages 58–67, January 1976.

[144] James R. Low. *Automatic Coding: Choice of Data Structures*, volume 16 of *Interdisciplinary Systems Research*. Birkhäuser Verlag, Basel, 1976.

[145] David Mathews. ISETL distribution page, 1997. At http://www.math.purdue.edu/~ccc/distribution.html.

[146] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[147] Thomas Minka. PLE lecture notes—Python, 1997. At http://vismod.www.media.mit.edu/~tpminka/PLE/python/python.html.

[148] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[149] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[150] Donald L. Muench. ISETL—interactive set language. *Notices of the American Mathematical Society*, 37(3):276–279, March 1990. Review of the software package ISETL 2.0.

[151] Henry Mullish and Max Goldstein. *A SETLB Primer*. Courant Institute of Mathematical Sciences, New York University, 1973.

[152] Netscape Communications Corporation. Core JavaScript reference, 1999. At http://developer.netscape.com/docs/manuals/js/core/jsref/index.htm.

[153] Object Management Group. CORBA, 1999. At http://www.omg.org/.

[154] The Open Group. *The Single UNIX Specification, Version 2*. The Open Group, February 1997. Six-volume set, Document Number T912. Freely searchable on-line at http://www.opengroup.org/publications/catalog/t912.htm.

[155] O'Reilly & Associates, 1999. At http://www.perl.com/. Home page for Perl.

[156] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998. Also at http://www.scriptics.com/people/john.ousterhout/scripting.html.

[157] R. Paige and J.T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 58–71, January 1977.

[158] Robert Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, January 1986.

[159] Robert Paige. Real-time simulation of a set machine on a RAM. In R. Janicki and W. Koczkodaj, editors, *Proc. ICCI 89*, volume II of *Computing and Information*, pages 69–73. Canadian Scholars' Press, Toronto, May 1989.

[160] Robert Paige. Efficient translation of external input in a dynamically typed language. In B. Pehrson and I. Simon, editors, *Technology and Foundations: 13th World Computer Congress 94, IFIP Transactions A-51*, volume 1, pages 603–608. North-Holland, September 1994.

[161] Robert Paige. Viewing a program transformation system at work. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming: Proc. 6th International Symposium, PLILP '94, Madrid, Spain, September 1994*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer-Verlag, 1994. PLILP '94 was a joint symposium with the 4th International Conference on Algebraic and Logic Programming, ALP '94.

[162] Robert Paige. Future directions in program transformations. *Computing Surveys*, 28A(4), December 1996.

[163] Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code—a case study. *Journal of Symbolic Computation*, 4(2):207–232, August 1987.

[164] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[165] Robert Paige, Robert E. Tarjan, and Robert Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40(1):67–84, September 1985.

[166] Robert Paige and Zhe Yang. High level reading and data structure compilation. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 456–469, January 1997.

[167] Parallax, Inc. BASIC Stamps, 1997. At http://www.parallaxinc.com/.

[168] Toto Paxia. A string matching native package for the SETL2 language, March 1999. Available through electronic mail upon request to paxia@cs.nyu.edu.

[169] Raymond P. Polivka and Sandra Pakin. *APL: The Language and Its Usage*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[170] Enrico Pontelli. Programming with sets, 1999. At http://www.cs.nmsu.edu/~complog/sets/.

[171] G. Rossi and B. Jayaraman, editors. *Workshop on Declarative Programming with Sets*, number 200 in Quaderni del Dipartimento di Matematica. Università di Parma, September 1999. Also at http://www.math.unipr.it/~gianfr/DPS/papers.html.

[172] Gianfranco Rossi. Programming with sets, 1998. At http://prmat.math.unipr.it/~gianfr/sets/index.html.

[173] E. Schonberg and D. Shields. From prototype to efficient implementation: A case study using SETL and C. Technical Report 170, Courant Institute of Mathematical Sciences, New York University, July 1985.

[174] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.

[175] J. Schwartz. *Set Theory as a Language for Program Specification and Programming*. Courant Institute of Mathematical Sciences, New York University, 1970.

[176] J. Schwartz, S. Brown, and E. Schonberg. SETLA user's manual. In *On Programming: An Interim Report on the SETL Project* [177], pages 90–159. OrigInally appeared as SETL Newsletter No. 70 [186].

[177] Jacob T. Schwartz. *On Programming: An Interim Report on the SETL Project*. Courant Institute of Mathematical Sciences, New York University, revised 1975.

[178] J.T. Schwartz. Automatic data structure choice in a language of very high level. *Communications of the ACM*, 18(12):722–728, December 1975.

[179] J.T. Schwartz. Optimization of very high level languages—I: Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, 1975.

[180] J.T. Schwartz. Optimization of very high level languages—II: Deducing relationships of inclusion and membership. *Computer Languages*, 1(3):197–218, 1975.

[181] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[182] Randal L. Schwartz. *Learning Perl*. O'Reilly & Associates, 1993.

[183] Keith Schwingendorf, Julie Hawks, and Jennifer Beineke. Horizontal and vertical growth of the student's conception of function. In Harel and Dubinsky [100], pages 133–149.

[184] Scriptics Corporation. Scriptics: Solutions for business integration, 1999. At http://www.scriptics.com/. Home page for Tcl/Tk.

[185] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, fourth edition, 1999.

[186] NYU SETL Project. *SETL Newsletters*. Numbers 1–234, Courant Institute of Mathematical Sciences, New York University, 1970–1981 and 1985–1989.

[187] David Shields. BALMSETL user's guide (in brief). *SETL Newsletters* [186], No. 20, March 1971.

[188] Josh Simon. Michigan Terminal System, 1999. At http://www.clock.org/~jss/work/mts/.

[189] M. Sintzoff. Calculating properties of programs by valuations on specific models. *ACM SIGPLAN Notices*, 7(1):203–207, 1972.

[190] W. Kirk Snyder. The SETL2 programming language. Technical Report 490, Courant Institute of Mathematical Sciences, New York University, January 1990. Also at ftp://cs.nyu.edu/pub/local/hummel/setl2/setl2.ps.Z.

[191] W. Kirk Snyder. The SETL2 programming language: Update on current developments. Technical report, Courant Institute of Mathematical Sciences, New York University, September 1990. Also at ftp://cs.nyu.edu/pub/local/hummel/setl2/update.ps.Z.

[192] Lindsey Spratt. *Seeing the Logic of Programming With Sets*. PhD thesis, University of Kansas, 1996. Also at http://www.designlab.ukans.edu/~spratt/papers/PhD_Dissertation.dir/visual_logic_sets_progra.html.

[193] Steven Pemberton. A short introduction to the ABC language, 1999. At http://www.cwi.nl/~steven/abc/.

[194] W. Richard Stevens. *UNIX Network Programming*, volume 1. Prentice-Hall, Upper Saddle River, NJ, second edition, 1998.

[195] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[196] Aaron M. Tenenbaum. *Type Determination for Very High Level Languages*. PhD thesis, New York University, October 1974.

[197] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996. See also http://haskell.org/.

[198] United States Department of Defense. *Reference Manual for the ADA Programming Language, ANSI/MIL-STD-1815A-1983*. Springer-Verlag, New York, February 1983.

[199] Guido van Rossum. Python patterns - implementing graphs, 1998. At http://www.python.org/doc/essays/graphs.html.

[200] Guido van Rossum. Comparing Python to other languages, 1999. At http://www.python.org/doc/essays/comparisons.html.

[201] Guido van Rossum. Python language website, 1999. At http://www.python.org/.

[202] Julius J. VandeKopple. Private communication, 1999.

[203] Julius J. VandeKopple. The SETLS programming language, revised 1994. At ftp://cs.nyu.edu/pub/languages/setls/.

[204] Herman Venter. The Slim programming language, 1999. At http://birch.eecs.lehigh.edu/slim/default.htm.

[205] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, second edition, 1996.

[206] Richard Wallace. The A.L.I.C.E. nexus, 1999. At http://www.alicebot.org/.

[207] H.S. Warren, Jr. SETL implementation and optimization: A first look at SETL compilation—target code style. In *On Programming: An Interim Report on the SETL Project* [177], pages 54–68. Revision of SETL Newsletter No. 53 [186].

[208] Gerald Weiss. Recursive data types in SETL: Automatic determination, data language description, and efficient implementation. Technical Report 201, Courant Institute of Mathematical Sciences, New York University, October 1985.

[209] Gerald Weiss and Edmond Schonberg. Typefinding recursive structures: A data-flow analysis in the presence of infinite type sets. Technical Report 235, Courant Institute of Mathematical Sciences, New York University, August 1986.

[210] Chung Yung. Extending typed λ-calculus to sets. In *Proc. MASPLAS '97*, April 1997. Also at http://cs1.cs.nyu.edu/phd_students/yung/publications/slambda3.ps.gz.

[211] Chung Yung. EAS: An experimental applicative language with sets. In *Proc. MASPLAS '98*, March 1998. Also at http://cs1.cs.nyu.edu/phd_students/yung/publications/eas02.ps.gz.

[212] Chung Yung. *Destructive Effect Analysis and Finite Differencing for Strict Functional Languages*. PhD thesis, New York University, August 1999.