THE ADA/ED SYSTEM: A LARGE-SCALE EXPERIMENT IN SOFTWARE PROTOTYPING USING SETL

Philippe Kruchten and Edmond Schonberg
New York University
Courant Institute of Mathematical Sciences

251 Mercer Street
New York City, N.Y. 10012
USA

Table of Contents

Abstract

Ada/Ed† is the first officially validated translator for Ada°. Ada/Ed was developed as a series of executable prototypes of increasing refinement, starting with an executable model of preliminary Ada in 1979, and evolving in parallel with successive versions of Ada (1980, 1982, ANSI standard). These prototypes were written in SETL, a very-high level language well-suited for software prototyping and experimentation. In this paper we examine SETL as tool for prototyping in the light of the design and evolution of Ada/Ed, indicating how the use of SETL makes it possible to construct operational prototypes that are malleable enough to undergo large modifications in an

evolutionary manner.

* This work was supported by U.S. Army contract number DAAB027-82-K-J196 (CORADCOM - Fort Monmouth, N.J.) and by the Ada Joint Program Office.

° Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

1. Introduction

It is becoming common wisdom that rapid software prototyping is the best way of insure the correctness of software systems. The promises of formal verification methods have not yet been realized, at least on a scale of interest to industrial systems (anything greater than 1000 lines of code) and designers see an increasing need for EXECUTABLE DESIGNS i.e. prototypes that display the operational characteristics of an intended system (apart from its eventual efficiency) and that can be produced rapidly at a fraction of the effort involved in the final product.

This need is particularly pressing when the requirements that define the system are themselves subject to change, or can be expected to evolve in parallel with the design of the system. In such cases, there is no question of FORMALIZING THE REQUIREMENTS being that there is in general nothing to formalize yet. Rather, a technique is needed to produce working sketches of the system, whose functionality can then be demonstrated to its potential users, and that are flexible enough to respond to changes in requirements by rapid changes in design. Software written in mainstream languages (PASCAL, the ALGOLs, etc.) does not have this malleability: it can be said to be hard and brittle (if you try to bend it, it breaks). Very-high level languages, particularly if they have a rich set of abstract primitives, are much better suited for prototyping purposes. We report here on the use of SETL in a large scale prototyping experiment that culminated in the construction of the first validated translator for Ada.

Section 2 of this paper describes the objectives of the Ada/Ed project and the stages of evolution of the system over the years. Section 3 gives a brief overview of SETL as a wide-spectrum programming language. Section 4 examines in detail some design decisions present in Ada/Ed, and their impact in subsequent versions of the system. Section 5 describes our programming environment. Section 6 discusses the central problem of efficiency, and offers some conclusions on our experience.

2. The NYUADA Project

The NYUADA project started in 1978, as a small-scale study of optimization issues relating to Ada. It became clear at once that Ada was at the time very loosely defined [ICHBIAHa, ICHBIAHb] and that a more rigourous definition of the language was needed, if any meaningful study of compilation issues were to be undertaken. This led to a first EXECUTABLE SEMANTIC MODEL OF ADA consisting of about 2000 lines of SETL, which described the executable aspects of the language in the form of an interpreter for an ad hoc intermediate representation dubbed AIS (Ada Intermediate Source). In the following two months, an LALR parser was added that generated AIS for correct Ada programs only [BURKEa, BURKEb]. In the course of the following 6 months static semantic checks were progressively added, so that when the 1980 definition of Ada was released [UNITED80] Ada/Ed was a working model of most of its static and dynamic semantics. Several features which were clearly unimplementable in their Preliminary Ada description (e.g. discriminants, and subprogram derivation) were purposely left out, and added when the 1980 Standard appeared. On the other hand, the semantics of tasking, including RENDEZVOUS, task activation, ABORT and SELECT statements were faithfully described from the beginning [DEWAR80].

At that point it became clear that Ada/Ed could serve two purposes: as a prototype translator, and as a rigourous definition of Ada. The latter use is a consequence of

a) the compactness of the system (25,000 lines of SETL, including documentation)
b) the very abstract model chosen to represent the run-time environment (See Sec. 4.3 below)
c) the demonstrable agreement between Ada/Ed and the Ada Compiler Validation Capability (ACVC) tests, which constitute an additional de facto definition of the language.

The NYUADA effort has subsequently pursued both goals mentioned above. In the course of 6 months, Ada/Ed was adapted to reflect the 1982 Reference Manual, and was further modified in early 1983 to reflect the final changes leading to the ANSI standard document [UNITED83]. Ada/Ed was validated by the Ada Joint Program Office in April 1983. From the inception of the 1979 NYUADA project to the validation of 1983 ANSI Ada/Ed, only 100 man-months were expended.

The ultimate proof of usefulness of a prototype is of course the construction of a production system that uses the prototype as a model. The NYUADA group is currently working on a substantially more efficient Ada translator, written in C, and using the current Ada/Ed system as a design tool. We discuss below the status of this new effort.

3. An Overview of SETL

Ada/Ed is a translator-interpreter written in SETL. SETL is a very-high level language developed at New York University, whose basic constructs are those of the theory of sets. Detailed descriptions of SETL can be found in [DEWAR79, SCHONBERG81, DEWAR82]. We limit ourselves here to the most salient points, and to those features of the language which contributed most to our prototyping work.

SETL is an imperative, sequential language with assignment, weak typing, dynamic storage allocation and the usual atomic types: numeric types, booleans, strings, and generated atoms (gensyms). It is thus a relative of LISP, SNOBOL and APL.

The most important data types of SETL are sets, tuples and maps.

1. Sets in SETL have their usual mathemathical properties: they are unordered collections of objects of arbitrary types, containing no duplicate elements, and on which the usual operations are defined: union, intersection, difference, power set construction. Set constructors and set iterators are built-in control structures that operate on sets. For example:

```
{x IN S | C(x)}
```

   denotes the subset of S, all of whose members x satisfy the predicate C(x). Quantified expressions over sets describe common search constructs:

```
EXISTS x IN S | C(x)
```

   is a boolean expression whose value is TRUE if there is an element x of S that satisfies C(x).

2. Tuples in SETL are ordered sequences of arbitrary size, which are indexed by positive integers. The components of a tuple, as well as the members of a set, can be of arbitrary types, so that sets of tuples, tuples of sets, tuples of tuples of sets of integers, etc. can be constructed. Insertions and deletions from either end allow tuples to be used as stacks and queues, concatenation makes tuples akin to lists. Heterogeneous tuples of fixed length are often used in the manner of record structures in other languages. Tuple constructors and iterators are similar to set constructors and iterators. For example:

```
[x:x IN [2..100] | (NOT EXISTS y IN [2..x-1] | x MOD y = 0)]
```

   constructs the sequence of primes smaller than 100.

3. Maps in SETL faithfully reproduce the notion of mapping in set theory: a map is a set of ordered pairs i.e. tuples of length 2. The first components of these tuples constitute the domain of the map, the second components its range. The elements of the domain and range of a map can have arbitrary types: we can define maps whose range is a set of maps, maps from tuples to sets of strings, etc. Maps can be used as tabular functions, and the application m(x), where m is a map, can be evaluated or be the target of an assignment. Finally, mapping can be multi-valued and single valued, and the image set m{x} of an object x under the mapping m can be obtained and assigned to.

These familiar notions constitute the core of SETL. What distinguishes SETL from purely mathematical discourse is the requirement of executability. To guarantee that all valid SETL constructs are executable, only finite objects can be denoted, and they must be built explicitly before they can be used elsewhere (there is no lazy evaluation in SETL). In order to speak about infinite objects in SETL, standard loop and procedure mechanisms are provided, in a conventional syntactic framework akin to that of PASCAL, but without full block structure.

This conceptual simplicity is supported by a complex run-time environment, able to execute all primitives on sets, tuples and maps defined over arbitrary types. We can say that sets, tuples and maps are abstract data types for which the SETL processor has available a full implementation, freeing the user from the need to specify himself how they should be represented and manipulated on an actual computer.

4. Modelling the Semantics of Ada in SETL

We will now discuss some features of the static and dynamic semantics of Ada, and show how they have been described in the Ada/Ed system. Let us first make the following obvious remark: prototyping means sketching, and thus deciding what to leave out. A prototyping tool is useful to the extent that it allows us to show precisely what is being described in detail, and what remains to be fleshed out. Thus, the success of Ada/Ed will be best shown by indicating what common aspects of a production implementation were ignored, or modelled in an abstract way that did not commit itself to any specific low-level implementation.

4.1 Static Semantics: Naming Declared Entities.

The first question to be addressed by a language processor is that of relating declared program entities to their attributes. In Ada this problem is complicated by the requirements of separate compilation, whereby arbitrary qualified names can refer to objects defined in subunits of other compilation units. For this reason, declared

entities must be given internal names that are guaranteed to be unique over a program library, and not just the current compilation. Such unique names are constructed in Ada/Ed as strings of arbitrary length; for example, a variable X defined within package PACK in procedure MAIN is represented internally by the name 'MAIN.PACK.X'. The rules of the language specify that the names of library units must be distinct, which insures that the strings so constructed are also unique.

Attributes of declared entities are represented by mappings whose domain is the set of such unique names. The collection of all these mappings constitutes what is normally called the symbol table. In Ada/Ed, rather than speaking of the symbol table, we usually speak of several independent mappings whose domains overlap. For example the map TYPE_OF takes a unique name into (the unique name of) its declared type. The map OVERLOADS takes an overloadable entity E (literal, subprogram name, or entry) into the set of unique names which have the same source identifiers as E and are visible at the point of declaration of E.

This simple structure bypasses the standard tedious problems of symbol table organization, and in particular the low-level implementation of separate compilation. These problems are not particularly deep, but it is only after establishing the correctness of the general naming framework that one can choose safely the low-level symbol table structures. The abstract organization of the Ada/Ed symbol table was in fact modified repeatedly before taking its present shape. Some of these modifications amounted (conceptually) to replacing a set of tuples by a mapping defined on some component of each tuple. Such a modification would have represented a major upheaval if concrete data structures had already been chosen. In SETL, they were accomplished by a replacement of a few tokens.

4.2 Static Semantics: Name Resolution

Name resolution is the process of relating source identifiers to their unique names. The rules of visibility in Ada are complicated by the existence of packages and the possibility of overloading entities with the same name. In Ada/Ed name resolution is performed by means of a few objects that reflect the declarative structure of a program. For each scope in which entities can be defined (subprogram, package, record, task, block, private part) a mapping is constructed, which takes a source identifier into the unique name assigned to it. A global map called DECLARED takes the unique name of each scope into the corresponding local mapping. Thus the range of DECLARED is a set of maps:

```
DECLARED: (scope_name → (identifier → declared entity))
```

In the case of packages, a distinction must be made between objects that are known within the package itself, and those that are known to other units. This distinction is reflected in the existence of a map called VISIBLE, whose domain is the set of visible package names, and whose image for any package is a subset of what is DECLARED for that package. For non-overloadable entities, the process of name resolution delivers the unique name corresponding to an occurence of an identifier I at location P in the source program. For overloadable entities, name resolution yields the set of possible meanings of this identifier at P. The full description of the state of the compilation requires in addition a stack of open_scopes, to indicate the current nest of lexical scopes available, and a set of used packages. All rules of visibility are correctly described by the two SETL procedures: find_simple_name, and collect_imported_names, whose text is given in full in Appendix 1.

The simple structure of DECLARED and the code for these procedures describe faithfully the rules of name resolution and visiblity. What is being sketched here is a series of hash-tables maintained across compilations, together with some complex rules for searching names in various files and directories. Once again, these low-level details are easy to handle once the logical structure of the system has been proved correct; too early a concern with these details will pollute the design with irrelevant considerations and often complicate the design task immensely. Conversely the simplicity of the structure of Ada/Ed has allowed us to find efficient algorithms for overload resolutions [SCHONBERG82].

4.3 Dynamic Semantics: Memory Management

Naming entities is the first problem that must be addressed by a model of the static semantics of Ada; allocating them is the first concern of a model of the dynamic, or run-time semantics. The storage model of Ada/Ed describes storage allocation without making any assumptions about the physical organization of memory: an allocation scheme is simply a mapping from unique names of entities to abstract CELLS called locations. These cells are obtained from some pool of indefinite size, and denoted by unique tags generated for each cell. This mapping is called EMAP (for environment map) and the semantics of each executable statement are described by their interaction with the current EMAP. The actual values denoted by each entity E are related to the current location of E by means of a global mapping CONTENTS. Thus, the value of E in the current environment is obtained by means of the composite application:

CONTENTS(EMAP(E))

and can be modified by assigning to the same construct. This model is similar to the model used in denotational language definitions. It allows us to treat correctly questions of aliasing, while avoiding all details regarding actual storage management: stack vs. heap, flat vs. segmented address space, etc.

In order to describe the dynamic semantics of scopes, a tuple called ENV_STACK serves to stack the environments of each currently active block. Each element of ENV_STACK contains its ancestor as a subset: in this fashion it is only necessary to refer to the top of ENV_STACK to retrieve the location of any visible entity. Thus the choice between displays and static chains to describe access to non-local entities is elided.

Finally such a stack of environments exists for each active task; this stack is accessed by means of a mapping from task names to environment stacks. The name of the task currently executing is used at all times to retrieve the environment in which it is executing. This simple mechanism avoids the machine-dependent details of context-switching, while making explicit in the text those portions of the run-time environment that must be critical sections.

4.4 Dynamic Semantics: The Organization of the Interpreter

One of the salient features of Ada is the specification of all those computations that may lead to the raising of an exception. The proper description of control flow during program execution must thus reflect the possibility that an exception may be raised at almost any point, terminating execution of the current frame, and possibly propagating across task boundaries.

The individual statements executed by the interpreter are high-level statements, corresponding nearly one to one with the statements and declarations of the Ada source. To each of the environments mentioned above is attached a STATEMENT STACK that represents at any time what is still to be done in that environment. Execution of a given task in a given environment consists then of removing the statement at the top of the statement stack and executing it. In order to model the flow of control, all departure from strict sequential execution is done by modification of this statement stack; for instance a LOOP statement is executed by pushing the loop again on top of the stack, then pushing the body of the loop, and then executing this modified statement stack; execution of an IF statement consists of evaluating of the condition and then pushing only the chosen branch on top of the stack (see Appendix 2).

Using the same mechanism, the more complex instructions are executed by pushing sequences of simpler ones, down to the level of atomic or uninterruptible elementary instructions. For example, execution of the assignment statement consists in the generation of three simpler instructions, which are pushed on the statement stack:

a) Evaluation of the name on the left-hand side
b) Evaluation of the expression on the right-hand side.
c) Actual data assignment

The execution of a) and b) leave both results on an EVALUATION STACK local to each environment. This decomposition indicates that the assignment statement is not indivisible, and that in fact an exception may be raised during the execution of any of its sub-statement. A similar decomposition is described for other statement forms. This decomposition can also be seen as on-the-fly code generation, and thus the Ada/Ed interpreter combines the functions of an interpreter and a code generator.

Instructions such as GOTO'S and EXIT'S are slightly more delicate: the statement sequence that follows a label is attached to the label name in the environment map. Executing a jump instruction consists of replacing the current statement stack by the statement sequence found in the environment map entry for the target label.

To handle exceptions, in addition to the statement stack just described, there is in each environment an exception handler. Whenever an error condition is encountered, the current statement stack is replaced by the content of the exception handler. If the exception handler is empty, or does not handle that specific exception, the current environment is discarded and the same exception is re-raised in the previous environment now on top of the environment stack.

This mechanism is very close to the continuation mechanisms found in denotational specifications of the language semantics; it provides a simple and concise way to describe Ada statements, but is of course extremely inefficient, due to the fact that statement sequences are copied again and again on top of the statement stacks to achieve iteration.

5. Prototyping-in-the-small versus Programming-in-the-large

It is now a commonplace in software engineering to distinguish between programming in the small and programming in the large. The former refers to projects involving few people working in close contact, while the later describes substantial projects requiring specific managerial machinery, protection mechanisms to isolate independent fragments of the project from each other, etc. It is also generally agreed that the former benefits from progress in programming languages, while the later depends more critically on the development of rational bureaucratic procedures.

A prototyping tool like SETL does not contribute very much to programming-in-the-large: it lacks the management layer mentioned above. But the compactness of SETL code vis-a-vis of other design tools allows one to regard as a small-scale problem what in an other environment would be considered a major effort, thus extending the range of problems that can be attacked with "programming-in-the-small" methods. If we assume that a skilled individual can retain intellectual mastery over an evolving system with 10,000 lines of code, then the efforts of 3 individuals suffice to guide the design and development of Ada/Ed. This is below the critical size that would

require the imposition of formal management procedures. Prototyping is by nature an activity to be carried "in-the-small", because it assumes rapid changes that cannot be accomplished when the number of participants grows too large. Thus the use of SETL allows us to regard Ada/Ed as a suitable project for prototyping-in-the-small.

This crucial advantage of a very-high level language such as SETL also offsets the fact the programming environment it offers is spartan, by today's standards. SETL has minimal support for separate compilation (which is less critical if the code to be recompiled is of moderate size) and no interactive debugging facility. The rest of the environment is the one provided by the VAX/VMS operating system, together with a rudimentary configuration management system built on top of the VMS file system. A good separate compilation facility, and a smooth way to interrogate the run-time environment (such as the one available on APL) would make SETL into a more polished prototyping tool.

Finally, there is something else one needs, which does not really belong to the programming environment: a group of naive users, that is people who make extensive use of the prototype, but do not know anything of its internal design. This provides the prototyping team with a continuous flow of bug reports in areas they would not have thought to look for, and this has substantially shorten the final integration phase. In our case, the ACVC team, led by J. B. Goodenough (at Softech) has developed the Validation Suite using our prototype, and therefore has been our most important user group so far.

6. Conclusions

The validation of Ada/Ed marks the end of the prototyping phase of the NYUADA project. We are now constructing an efficient Ada system modelled after the Ada/Ed prototype. The new system is being constructed in two steps: a first version in SETL followed by the final one in C. The new SETL version addresses all design details that are described abstractly (or not at all) in the prototype, and lowers the semantic level of most of the SETL code present in Ada/Ed (for example by introducing integer domains instead of strings, replacing sets by lists where possible, etc.).

An additional facility of SETL bridges the gap between weakly-typed code and strongly-typed code [DEWAR79]. It is possible to annotate a SETL program with declarations that specify concrete data-structures for sets and maps. For example, one can require that some sets be represented internally as bit-vectors, others as lists, some mappings realized as arrays, etc. These optional declarations can increase the efficiency of a SETL program; they also serve to bring the SETL program to a more concrete level, from where its transcription into an efficient system language is an almost mechanical procedure.

We are now performing the first step of this conversion for the front-end and semantic phase of Ada/Ed, namely rewriting it into low-level SETL. We anticipate that 6 man-months will be sufficient to complete this step. The transformation of the Ada/Ed interpreter is a more complex affair, because the current interpreter is a model both of the run-time environment and of a code generator for Ada. Accordingly we have been working on the design of the code generator, using the current interpreter as prototype, and on the comparatively simple design of a low-level interpreter of conventional nature.

Thus Ada/Ed is the design document for our next system [NYUADAa, NYUADAb]. As was pointed out in the introduction, its usefulness consists not only in giving detailed descriptions of many parts of the system, but also in indicating plainly where the problems lie and what remains to be designed. It also serves as ongoing documentation, as a readable introduction to the system for new members of NYUADA, and as an informal arbiter on semantic questions (we often resolve a query about the meaning of an Ada construct by examining the corresponding SETL code in Ada/Ed).

It must be noted that additional gains follow from the use of a software prototype, in those cases where the prototyping language and the final implementation language are one and the same, or closely related (as is the case for our low-level SETL version); namely, portions of the system that are not efficiency-critical can be re-used directly.

This indicates the interest of wide-spectrum languages in which a number of semantic levels are available, and where the transition from prototype to production software can take place within the same linguistic frame. When such an integrated framework is available, there is every reason to produce a FULLY FUNCTIONAL PROTOTYPE because every phase of development of the final product will be testable, and a large fraction of routine boiler-plate code (which otherwise would not justify prototyping) may not have to be written more than once.

Ada/Ed could not have been constructed without the facilities of SETL. The success of the project must also be credited to a particular way of using SETL, namely writing in a very abstract programming style, and making a deliberate effort to ignore efficiency questions until after all semantics questions of Ada had been fully resolved. The slogan SLOW IS BEAUTIFUL was used as an incantation by members of the project, in order to keep our attention focused in this fashion; this slogan may be the best capsule summary of the practice of software prototyping.

Acknowledgements

Ada/Ed is the work of a number of individuals. Its first architects were R. B. K. Dewar and G. Fisher. R. Froehlich designed large portions of earlier versions of the interpreter. The system was brought to completion thanks to the efforts of B. Banner and Ph. Charles. S. Bryant, M. Burke, C. Goss, and T. Seisser contributed valuable components of the system. Our gratitude to SETL, to its designer J. T. Schwartz, and its implementors, in particular D. Shields and S. Freudenberger, can bear restating: without SETL none of our work would have been possible.

References

BURKE, M. G., AND G. A. FISHER JR.:
A Practical Method for Syntactic Error Diagnosis and Recovery.
Proc. of the SIGPLAN'82 Conference on Compiler Construction, Boston, Mass. June 1982, ACM (1982a).

BURKE, M. G.:
A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery.
(Unpublished Ph. D. dissertation)
New York: Courant Institute of Mathematical Sciences, NYU, December 1982 (b).

DEWAR, R. B. K., A. GRAND, S. C. LIU, J. T. SCHWARTZ, AND E. SCHONBERG:
Programming by Refinement, as exemplified by the SETL Representation Sublanguage.
ACM Trans. Program. Lang. Syst. 1(1), 27-49 (July 1979).

DEWAR, R. B. K. ET AL:
The NYU Ada Translator and Interpreter.
Proc. of the Compsac'80 Conference, IEEE, Chicago October 1980.

DEWAR, R. B. K., E. SCHONBERG, AND J. T. SCHWARTZ:
High-Level Programming - An Introduction to the Programming Language SETL.
New York: Courant Institute of Mathematical Sciences July 1982.

ICHBIAH, J. D.:
Preliminary ADA Reference Manual.
SIGPLAN Notices. 14(6), Part A (June 1979a).

ICHBIAH, J. D. ET AL:
Rationale for the Design of the ADA Programming Language.
SIGPLAN Notices. 14(6), Part B (June 1979b).

NYUADA:

Semantic Actions for ADA.

(Technical Report #84)

New York: Courant Institute of Mathematical Sciences 1983(a).

NYUADA:

An executable Semantic Model of ADA.

(Technical Report #85)

New York: Courant Institute of Mathematical Sciences 1983(b).

SCHONBERG, E., J. T. SCHWARTZ, AND M. SHARIR:

An Automatic Technique for Selection of Data Representations in SETL Programs.

ACM Trans. Program. Lang. Syst. 3(2), 126-143 (April 1981).

SCHONBERG, E., AND G. A. FISHER JR.:

An efficient Method for Handling Operator Overloading in ADA.

Proc. of the AdaTEC Conference on ADA, Arlington, Va. October 1982, ACM, 107-111 (1982).

UNITED STATES DEPARTMENT OF DEFENSE:

Reference Manual for the ADA Programming Language.

Proposed Standard Document, July 1980.

UNITED STATES DEPARTMENT OF DEFENSE:

Reference Manual for the ADA Programming Language.

ANSI/MIL-STD-1815 A, January 1983.

Appendix 1: SETL procedures for name resolution in Ada/Ed

```
PROCEDURE find_simple_name(id);

  IF EXISTS sc = OPEN_SCOPES(sc_num) |
      is_defined(u_name := DECLARED(sc)(id)) THEN
    IF not can_overload(u_name) THEN
      found := u_name ;
    ELSE
      names := OVERLOADS(u_name) ;

     $ Scan open scopes for further overloadings.
      (FOR I IN [sc_num+1..#open_scopes])

        IF is_defined(u_n := DECLARED(open_scopes(i))(id)) THEN
            CONTINUE FOR I ;

        ELSEIF NOT can_overload(u_n) THEN
            found := names ;

        ELSE names +:={o in DECLARED(u_n) |
                    NOT EXISTS n IN names |
                    (same_type(type_of(n),type_of(o)) AND
                    same_signature(signature(n),signature(o)))};
        END IF ;
     END for ;
     imported := collect_imported_names(name) ;

    $ Keep only the imported names which are not hidden
    $ by visible names with the same type and signature profile.
      IF is_overloaded(imported) THEN
        names +:= { foreign : foreign IN imported |
                  NOT EXISTS n IN names |
          same_type(type_of(n),type_of(foreign)) AND
          same_signature(signature(n),signature(foreign))} ;
      END IF ;

      found := names ;
    END IF ;

  ELSEIF (imported := collect_imported_names(name))/={} THEN
    found := imported ;
```

```
  ELSE $ Undefined identifier, error message will be emitted.
    found := '?' ;
  END IF ;

  RETURN found ;
END PROCEDURE find_simple_name ;

PROCEDURE collect_imported_names(id) ;

$ This procedure collects the set of all imported names corresponding
$ to identifier -id- which appear in currently visible packages.
$ An imported identifier is visible if:
$ a) It is not an overloadable identifier, and it appears in only
$    one used package.
$ b) Or else all of its appearances in used packages are overloadable.

  imported := {};
  (FORALL used IN USED_PACKAGES | is_defined(f:=visible(used))
                        AND id_defined(foreign:=f(id)))
    IF can_overload(foreign) THEN
      imported +:= overloads(foreign) ;
    ELSE
      imported WITH := foreign ;
    END IF ;
  END FORALL ;


  IF EXISTS fgn IN imported | NOT can_overload(fgn) THEN
    $ If it is the only name found, return it.
    IF #imported = 1 THEN
      RETURN fgn ;
    $ else various visible names hide each other.
    ELSE
      RETURN {} ;
    END IF ;
  ELSE    $ all occurences are overloadable.
    RETURN imported ;
  END IF ;

END PROCEDURE collect_imported_names ;
```

The text above (taken verbatim from Ada/Ed) makes use of several self-explanatory predicates:
is_defined, is_overloaded, same_type, same_signature, can_overload. The operator "+" is overloaded in SETL, and denotes among other things set union. The tuple OPEN_SCOPES reflects the nest structure of blocks at each point in the compilation process. The set USED_PACKAGES holds the unique names of the packages that appear in currently applicable USE clauses.

Appendix 2: examples of Ada statements modelling: loop et if

```
$------
$ ['loop',body]
$
$ The loop statement works by duplicating the body of the loop ahead of itself
$ on the statement stack each time the loop is executed. Note that
$ this correspond to a loop without a label.

('loop'):
 [-,body]:= STM;
 exec(STM);
 exec(body);

$------
$ ['if',cond_list,elsebody]
$
$ The if statement contains a tuple of condition-body pairs of the form
$ [condition,if_body], where condition is a boolean valued expression and
$ if_body is the sequence of statements to be executed if that condition is true.
$ If an else part is present, the else_body is executed if the condition
$ is false and there is no more condition_body to evaluate.

('if'):
  [-,cond_list,else_body]:=STM;
  IF cond_list/=[] THEN
    [cond,body] FROMB cond_list;
    exec([['veval_',cond],
        ['if_',body,[['if',cond_list,else_body]]]]);
  ELSEIF present(else_body) THEN
    exec(else_body);
  END IF;

$ ['if_',true_body,false_body]
$
$ This as a subsidiary statement to the 'if' above; it takes the
$ boolean value on top of the value stack, and executes the proper
$ body, by copying it on top of the statement stack.

('if_'):
  [-,true_body,false_body]:=STM;
  pop(value);
```

```
test_value:=(boolean_true=value);
IF test_value AND (true_body/=[ ]) THEN
  exec (true_body);
ELSEIF NOT test_value AND (false_body/=[ ]) THEN
  exec (false_body);
END IF;
```

In those two fragments of the main CASE OF in the Ada/Ed interpreter, STM denotes the current statement, which is first unpacked; the exec procedure pushes a sequence of statements on top of the statement stack: pop pops a value from the value stack; the subsidiary statement 'veval_' calls the expression evaluator.