CHAPTER IV

Interaction of Programming Language and Environment Design

A programming language and its program development environment (which includes, for example, components to create, modify, verify, compile, link, execute, and debug programs) are interrelated in the sense that the ease with which programs can be developed depends upon the facilities provided by the environment, which in turn depend upon the constructs defined in the programming language. For example, a structure editor is a useful tool to develop programs, but only if the programming language is block-structured.

This chapter investigates some of the influences that programming languages and editing environments have on each other. We'll start by studying the effect of programming languages on environments, and then explore the influence of environments on languages. In particular we'll examine a RESOLVE editing environment designed to execute on Macintosh personal computers, and some of the ways its design influenced RESOLVE's design.

4.1 Programming Language Influence on Environmental Design

Perhaps the biggest impact programming languages have had on editing environments in recent years has been the development of structure and syntax directed editors. Using a structure editor, the document being edited is considered to be a structure (e.g., a tree), not just a sequence of characters. Moving around the document is accomplished with commands such as "move to next sibling" or "move to parent." Creation and modification of a document is done by modifying the document's structure. A structure editor is useful for editing any structured document, such as programs and manuscripts. Mentor [Donzeau-Gouge 84a, Donzeau-Gouge 84b] and Tioga [Teitelman 85, Swinehart 86] are examples of structure editors.

Syntax directed editors extend structure editors in that the structure imposed on the document is the structure of some programming language [Meyrowitz 82]. In other words, syntax-directed editors are structure editors that enforce (or at least have knowledge of) a particular programming language. The Cornell Program Synthesizer [Teitelbaum 81], Gandalf [Habermann 86] (including GNOME [Garlan 84] and GENIE [Chandhok 87, Chandhok 88]), and PECAN [Reiss 85] are examples of syntax directed editors.

Syntax directed editors have several potential advantages over traditional text editors. First, an editor can be designed in which it is impossible to create syntactically and static-semantically incorrect programs, thus reducing errors. Second, the editor can automatically take care of many of the mundane syntactic structures, such as keywords and "syntactic sugar", allowing more rapid program construction. Finally, the compiler is simplified if the editor produces an error-free parse tree; in fact, it is possible to have the editor translate the parse tree as it is created, eliminating the need for a separate compiler.

Despite the advantages of syntax directed editors, some researchers (e.g., [Waters 82]) argue that editors should have both structure and text capabilities. The reasons cited are that many programmers prefer to edit text, and that some editing (e.g., entering in-fix expressions and transforming an if-statement into a while-statement) are significantly easier to do by editing text. Others (e.g., [Cohen 82] and [Shani 83]) argue that the problems (if any) are in the existing structure and syntax directed editors, not the approach, and[Garlan 84] states that "students, as a whole, have *not* found the structure/infix mode of entry for expressions to be a problem" in referring to the GNOME environments. Also, the ability for "free typing" complicates the editor because it must include a parser to determine the structure of newly-entered text and insert it into the program structure; in addition the editor must specify what happens to text that is incorrect. Nonetheless, most structure and syntax directed editors include some ability for "free typing."

Although it is possible to develop a syntax directed editor for any programming language, they are most useful for "block structured" languages such as Pascal, Algol, and Ada. For instance, a syntax directed editor for BASIC would not be very useful because BASIC programs are organized as a sequence of statements rather than a tree.

This suggests an important influence that a programming language has on its environment — whether or not a syntax directed editor is an appropriate tool for creating and modifying programs.

There are other ways that a programming language influences its environment. For instance, the hardware platform on which an environment executes is affected by the language's vocabulary. For example, environments that rely on standardized character sets for input and output (e.g., ASCII terminals) are not well suited for languages that use special symbols (e.g., APL).

Programming languages have influenced the design of machine architectures. Whereas early architectures were optimized for machine and assembly language programming, modern ones are designed for efficient implementation of programs coded in high level languages.

Also, a language that encourages development of separately-compiled modules (e.g., Ada, Eiffel, CLU, and RESOLVE) suggests that the environment include a component to help manage the module library. The librarian aids the programmer in selecting the appropriate module, and/or validates module use.

In fact, programming environments have evolved to a large extent in response to new developments in programming language design.

4.2 Environmental Influence on Programming Language Design

Historically, program development and execution environments have significantly influenced the design of programming languages. For example, the format of FORTRAN statements is directly related to the editing environment of the day — 80-column punch cards. Also, many of FORTRAN's control statements (e.g., the computed IF) are included in the language because of analogous statements in the machine language of the expected target computer, the IBM 709 [MacLennan 83]. The goal of a simple one-pass compiler influenced the design of Pascal [Wirth 83], and more recently Ada includes only constructs whose implementation is clear and efficiently implementable using known techniques [DoD 83][36].

---

36 This demonstrates that terms such as "clear," "efficient," and "known" are relative!

However, programming environments have not had as much influence on programming languages as programming languages have had on environments. It seems that often a language is completely designed before significant consideration is given to developing an environment for it, and therefore environmental concerns have little or no impact on the design of the language.

RESOLVE was designed concurrently with an editing environment, which influenced the syntax of the language. This editor is designed to meet several goals. For example, it is designed to execute on Macintosh personal computers, and takes advantage of a mouse and menus for syntax directed editing, makes minimal use of the keyboard (i.e., the keyboard is used only to name identifiers, and there is no option for "free typing" program text), and operates within a relatively small monochrome bit-mapped display. Also, the editor is responsible for processing several syntactic constructs that are usually defined in the syntax of contemporary programming languages, such as operator overloading and infix notation. (Portions of the editor have been implemented, and are described in Appendix A.)

Let's take a closer look at the editor's influence on the syntax of RESOLVE. First, keywords are inserted into a program using menus, and are never entered using the keyboard. Therefore, keywords in RESOLVE are not abbreviated because abbreviated keywords have no advantage over unabbreviated ones with respect to the ease of program construction, yet are potentially cryptic and confusing.

Second, because the display is narrow and the layout of the display is controlled by the editor (e.g., the programmer cannot insert carriage returns randomly in the program), RESOLVE's syntax is designed so statements do not become arbitrarily wide. For example, formal parameters are defined in a parameters section following the heading rather than within parentheses in the heading. Also, actual parameters are restricted to be identifiers, rather than permitting function invocations with potentially long actual parameter lists.

Finally, RESOLVE's syntax does not include facilities for complex syntactic structures such as overloading and infix notation. Instead, these are handled by the editor using its "alternate syntax" feature. Operation invocations normally appear in a prefix syntactic form — the operation name is followed by the actual parameters enclosed in parentheses. Though this is adequate, some constructs are visually unattractive. For

example, the usual assignment statement "`z := x + y`" is displayed as "`z := Add(x,y)`".

The RESOLVE editor allows any operation to have alternate syntax, which defines the format of all invocations of that operation. The alternate syntax for an operation is a sequence of characters which must include every formal parameter of the operation. The editor uses the alternate syntax definition as a template, substituting actual parameters for the formals in the alternate syntax definition.

For example, function Add defined in conceptualization Bounded_Integer_Template (see Figure 21 in Section 3.6.1) has two formal parameters — i and j — so every alternate syntax definition for Add must include i and j. Figure 26 demonstrates the effect of alternate syntax by showing several alternate syntax definitions for Add and the way the editor displays an example invocation.

Alternate Syntax	Example Invocation
(no alternate syntax)	z := Add(x,y)
"i + j"	z := x + y
"i j +"	z := x y +
"sum of i and j"	z := sum of x and y

Figure 26

Effect of Alternate Syntax

It is important to understand that alternate syntax only affects the way an invocation is *displayed* by the editor. Alternate syntax does not change the way a programmer inserts an operation invocation into the program — she or he still uses the mouse and popup menus (see Appendix A).

This raises an interesting concern, though — what if the alternate syntax of two operations results in a program whose textual display is ambiguous? For example, if functions Add and Subtract both have "`i + j`" as alternate syntax, it is not clear from the display of a program whether the statement "`z := x + y`" is an invocation of Add or Subtract.

From the editor's viewpoint this ambiguity is not a problem. When the "ambiguous" statement was constructed the programmer explicitly indicated (via a popup menu) which operation to invoke, which is the operation inserted into the parse tree. In other words, the parse tree is constructed by the programmer using the editor; it is not created by parsing the textual form of a program. In fact, because the editor does not permit a program to be entered by "free typing," the textual form of the program is never parsed, and the fact that it might look ambiguous is not relevant for processing by the editor or compiler.

For obvious reasons of clarity it is not recommended that programmers define alternate syntax that might allow a program to be developed whose display is ambiguous. Therefore, when alternate syntax is defined the editor checks it for conflict with other definitions. If a conflict is detected, the programmer is appropriately warned, but is not prohibited from making the ambiguous definition.

In addition to permitting very general operator overloading, the alternate syntax facility can be used to display a procedure invocation to reflect its effect. For example, defining "`x :=: a[i]`" as alternate syntax of procedure Access in conceptualization Array_Template (see Figure 20 in Section 3.5.3) causes the invocation "`Access(arr,index,z)`" to be displayed "`z :=: arr[index]`", which accurately reflects the effect of the invocation as an apparent swap statement in which an array element acts like a variable.

Alternate syntax appears to be a very powerful syntactic mechanism, allowing the editor to effect syntactic constructs such as overloading and notation without complicating program parsing or the compiler.

4.3 Summary

In this chapter we've briefly explored the interaction between programming language design and program environment design. Some of the influences that programming languages have had on environments in recent years include structure and syntax directed editors, hardware platform designs for both program development and program execution, and the development of environment components such as librarians.

Although program development and execution environments influenced the design of early languages such as FORTRAN, their influence recently has been minimal. In fact it seems that environments are often developed after the programming language is designed. However, an editing environment was designed concurrently with RESOLVE, and influenced the language in several significant ways. In particular, RESOLVE keywords are not abbreviated, parameter lists are such that statements do not become arbitrarily wide, and several complex syntactic constructs such as operator overloading and infix notation are handled as alternate syntax by the editor rather than complicating the syntax of the language.

RESOLVE is an example of a programming language whose design was influenced in positive ways by the environment, and supports the third point of the thesis.