A language-independent programmer’s interface

by ROBERT M. BALZER
USC/Information Sciences Institute
Marina del Rey, California

INTRODUCTION

This paper addresses the general problem of creating a suitable on-line environment for programming. The amount of software, and the effort required to produce it, to support such an on-line environment is very large relative to that needed to produce a programming language, and is largely responsible for the scarcity of such programming environments. The size of this effort was largely responsible for the scrapping of a major language (QA4) as a separate entity and its inclusion instead as a set of extensions in a LISP environment. The few systems which do exist (e.g., LISP, APL, BASIC, and PL/I) have greatly benefited their users and have strongly contributed to the widespread acceptance of the associated language.

At a bare minimum, a suitable programming environment consists of an on-line interpreter (or incremental compiler), an integrated interactive source-level debugging and editing system, and a supporting file structure. More extensive environments would include such facilities as automatic spelling correction, structural editors, tracing packages, test case generators, documentation facilities, etc.

Looking at several programming environment systems, one recognizes much uniformity. Most of the software supporting these systems is similar in both its organizational structure and functions. The systems differ in detail more from style differences between the system designers than from differences required by the programming languages.

The Programmer’s Interface (PI) concept attempts to exploit this uniformity by creating a single programming environment capable of easily interfacing users with a wide variety of on-line programming languages. Users would then have the full facilities of this environment at their disposal. The PI is thus responsible for transforming these programming LANGUAGES into SYSTEMS. The cost of providing such an environment for a language would drop from the several man-years now required to the few man-days (estimated) to interface to a PI. Additionally, the existence of a common programming environment for many different languages would justify the inclusion of further capabilities.

This common programming environment provided by a PI should include facilities for: creating, modifying, storing, and retrieving programs; on-line debugging, including trace and break facilities as well as the facilities of the language for evaluation of expressions at breaks; modifying the interface between routines (via an ADVISES capability); automatic spelling correction; remembering, modifying, and reissuing previous inputs; and undoing the effects of any of these PI facilities.

Such a PI has been constructed and interfaced to the programming language ECL.7 The remainder of this paper explains the PI concept in terms of this implemented program. The deficiencies of this particular implementation are discussed in the conclusion.

SYSTEM ARCHITECTURE

The facilities provided by the implemented Programmer’s Interface (PI-1) are based on the INTERLISP (formerly BBN-LISP) system. In fact, they are the facilities of this system, as modified for language independence. The Programmer’s Interface itself is implemented in INTERLISP and coexists with the facilities it invokes to provide the programming environment. INTERLISP was chosen as the basis both because it already had an extensive set of programming tools in an accessible form, and because their structure and operation could easily be altered to operate as required for a PI.

The system structure is shown in Figure 1. The ARPA Network is used as the communications mechanism between PI-1 and the user’s language processor. This choice has three advantages. First, it allows the interfacing of PI-1 to any language processor available on the ARPANET independent of what machine it runs on. Second, this interfacing can be done by PI-1 without the knowledge of the language processor. Thus no modifications to the language processor are required. Finally, the use of the Network greatly simplifies implementing the interconnection by allowing external character strings to be used for communication, rather than internal data structures with the attendant incompatibility problems.

Three properties are required of a language processor for its use with a PI:

1. There is a way to form a coroutine linkage between the language processor and the PI by interconnecting their I/O ports. This type of linkage is discussed in
If the user’s input is intended for his language processor, it is passed across the ARPA Network to that language processor. Any output generated by the processor is received across the Network again by PI-I. It suppresses the echo of the input and passes the output to the user, extracting from it the “value” and putting it into the history list for use by the Programmer’s Assistant.

If the user’s input is an environment-type command and should be performed within PI-I, the appropriate facility is invoked. In simple cases the operation completes, returns a value that is put in the history, and another input is processed. In more complex situations, some interaction is required with the user’s language processor. This is accomplished by dynamically generating a series of inputs for the language processor that will have the desired effect or return the desired information. These are passed through the communications mechanisms to the processor; its output is captured; and either the success of the modifications is verified or the desired information is extracted. Any number of such cycles may be required before the PI-1 facility completes its processing of the user’s command. As an example, considered the loading of a file. As the function definitions are read in, they are stored as a property of the corresponding atoms to be used by the PI-1’s editor for any modifications required later. The function definitions also are passed to the language processor so that it can use these for evaluation. Thus, one cycle is required for each function defined in the file.

PI-1 maintains a copy of all functions defined by the user and this is used by PI-1’s editor when the user alters the definition. Whenever this definition changes (by redefinition or through exiting the editor), the resulting definition is passed to the language processor as a new definition of the function.

INTERFACING A LANGUAGE TO A PROGRAMMER’S INTERFACE

Most of PI-1 is language independent, but certain portions must be modified to accept a new language. These fall into the categories of syntax modification, synchronization, program writing, and debugging.

The INTERLISP editor used by PI-1 is structural rather than string-oriented. To be effective, the text it is manipulating must have a structural basis. The syntax modification routines are responsible for introducing the structure into the user’s language (only for use within PI-1). This structure is of two forms. First is the grouping of characters into lexical units. The user’s language may have very different lexical grouping rules than LISP and the syntax modification package is responsible for the lexical analysis. Second, the lexical units thus produced are grouped into larger units by the use of parentheses. These units can be nested within one another to form the familiar LISP s-expression structure. The designer of the syntax modifier must decide where to introduce this structural grouping. In ALGOL-like languages,
a natural place would be to group the lexical units of a statement together and groups of statements within blocks together. The structural groupings selected are introduced into all program text input by the user, and used by him to direct the editor in its modifications of this text. When this text is passed to the language processor, those structural groupings artificially introduced for editing purposes are removed before transmission.

PI-1 and the language processor must be synchronized and kept in step with each other. Logically this is very simple and is accomplished by having PI-1 wait until the language processor has completed evaluating the previous input before giving it another. This situation is signaled by the language processor's attempt to read the next input. Unfortunately (due to a deficiency in the network protocol), this information is not available. Therefore the language processor's state of readiness must be determined by examination of its output stream. Fortunately, most on-line language processors explicitly indicate their readiness for more input by providing the user with a prompt character.

PI-1/ECL EXAMPLE

The following actual example indicates the use of PI-1 with the programming language ECL. The prompt character (as defined by ECL) is either →, *, or a number followed by :>. Commentary is enclosed in square brackets.

→ 3+4
7
→ TEST1←EXPR(A:INT,B:INT,INT)BEGIN A+B; END;

( TEST1)
 * TEST1(3,4)
 7
 → EDITF(TST1)
 = TEST1
EDIT
*PP

(EXPR(A : INT, B : INT ; INT)
 (BEGIN (A + B)
   END))

*F BEGIN P
(BEGIN (A + B) END)
*(A GT B ⇒ A−B; A+B))

*PP
(BEGIN (A GT B ⇒ (A − B))
 (A + B)
END)
*(-4 (A=B → B ← 2*A))

The language processor's output must be scanned for this prompt and this is used as a synchronization mechanism between PI-1 and the language processor.

Several facilities within PI-1, such as break, trace, and advise, cause additional statements to be written into the user's program for evaluation at runtime. The interfacer of a new language must specify the form of these additions.

PI-1 contains many advanced debugging capabilities not found in most language processors. These aids are all based on information gathered during execution or at a break point within the program. To use these facilities, the designer of the language interface must supply routines that provide the basic information on which these debugging aids are built.

PI-1 took approximately three weeks to implement and debug, including the language interface to ECL. Although no other language interfaces have yet been built, it is estimated that an interface to another suitable language could be designed, implemented, and debugged in less than a week.
•PP
  (BEGIN (A GT B => (A - B))
   (A + B)
   (A = B -> (B <- 2 * A))
   END)

•UNDO
  (-4 --) UNDONE.
•PP
  (BEGIN (A GT B => (A - B))
   (A + B)
   (A = B ~ (B ~ 2 * A))
   END)

•USE -3 FOR -4
•PP
  (BEGIN (A GT B => (A - B))
   (A + B)
   (A = B ~ (B ~ 2 * A))
   (A + B)
   (END)

•OK
TEST1
  → TEST1(3,4)
  7
  → TEST1(4,3)
  1
  → TEST1(4,4)
  12
  → ADVISE(TEST1 BEFORE (A<- 2*A))

TEST1
  → TEST1(3,4)
  2
  → USE 6 3 10 FOR 4
  18
  3
  16
  → ADVISE(TEST1 AFTER (VALUE\- VALUE-1))

TEST1
  → REDO USE

TYPE FAULT
  → BROKEN
NIL
TYPE FAULT
  → BROKEN
NIL
TYPE FAULT
  → BROKEN
NIL
3:: RETBRK(0)
NIL
  → TEST1(3,4)

[Prettyprint it.]

[User notices his error (addition made at wrong spot) and asks system to undo last command.]

[Check to see that it's really gone.]

[Substitute -3 for -4 in the insertion command and reissue it.]

[Make sure addition put in correct spot.]

[Exit editor.]

[Test function. A is less than B, just add them.]

[A greater than B, subtract B from A.]

[A=B, double B and add in A.]

[Modify TEST1 so that before it is entered, but after its parameters have been bound, the value of A is doubled.]

[Invoke modified function.]

[Double 3 to get 6 and subtract 4.]

[Successively substitute 6,3, and 10 for 4 in the last statement.]

[TEST1(3,6)]
[TEST1(3,3)]
[TEST1(3,10)]

[Modify TEST1 so that after it is finished, but before it returns, the value to be returned is decremented by 1.]

[Reissue the previous USE command (which generated the 3 invocations of TEST1)]

[3 type fault error occur.]

[Go back to top level.]

[Try simple case.]
A Language-Independent Programmer's Interface

Type fault — broken
NIL
1:> IN?
IN ENTRY OR EXIT OF...
VALUE\ \ ← \ VALUE ~ 1

1:> EDITF(TESTI)
EDIT
+F VALUE 0 P
(VALUE\ \ ← \ VALUE ~ 1)
*R VALUE VALUE\ \ 
*OK
TESTI
1:> TESTI(3,4)
(1)
1>
NIL
→ REDO USE
(17)
(2)
(15)

Conclusion

An extensive programming environment has been created for the ECL language through a program (PI-I) which allows the use of the already existing INTERLISP facilities. This greatly expands the user's facilities for creating, editing, and debugging his programs. His programming language has been transformed into a programming system. The availability of a comprehensive set of "environment" tools working in conjunction with the programmer's language is extremely important to his productivity.

The significance of this work, however, lies not in the particular interface provided between INTERLISP and ECL, nor in the extensive capabilities provided the user, but rather, in (1) the observation that very little of the interface itself, or of the capabilities provided, are language dependent, (2) the recognition that the programming environment can be effectively split into an "environment" part and an execution and evaluation part and (3) the experience gained from building such a system and interfacing a language to it.

PI-I, however, suffers from a number of deficiencies, the most important of which is the use of already existing tools in more general environments than they were designed for. This was most notable in the use of LISP's editor for non-structured text (and the need therefore to introduce structure by parentheses) and the need to replace LISP's input routines to provide the proper lexical analysis for the interface language. Both of these problems could be avoided in a PI by having it use the syntax description of the language to guide the input, and editing and display of programs.

One of the strengths of the PI concept is the split between the "environment" part and the evaluation part. This split, however, introduces the problem of communication and synchronization; each part must keep the other informed about changes it makes that affect the other. In PI-I, this communication and synchronization was partial and clumsy. The flow of information from the environment to the evaluation part was adequate, but the reverse flow was not. The need to communicate to another program suitable explanations of what the state of the evaluation was, what the cause of the error was, or even that an error occurred was simply not envisioned or planned for.

PI-I has thus demonstrated that a moderately integrated PI can be built that has facilities for beyond what is typically available at a fraction of the cost. However, development of highly integrated PI will have to await a better understanding of the functional requirements of a language processor in such an environment.

Although the Programmer's Interface has only been interfaced to one language (ECL), and although it only contains a small fraction of the capabilities ultimately desired, it is having a major effect by acting as a prototype for a major software project being undertaken to develop this understanding and provide a single, common, compre-
hensive programming environment interfaced to a wide variety of languages running on many different machines communicating through a network. New languages or machines could be interfaced to the system at a fraction of the cost of providing a separate programming environment. Widespread usage would justify the expenditure of more resources to augment and improve the capabilities provided. Such a PI could free users from having to develop their programs only with software available on their own machines and could provide a much more comprehensive and coordinated software development package than is currently available.

REFERENCES