AFDELING INFORMATICA IW 185/81 DECEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

J. HEERING & P. KLINT

TOWARDS MONOLINGUAL PROGRAMMING ENVIRONMENTS

Preprint

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1980 Mathematics subject classification: 68B20

ACM-Computing Reviews-category: 4.22, 4.35, 4.42

Towards monolingual programming environments†

by

Jan Heering & Paul Klint

ABSTRACT

Most programming environments are much too complex. One way of simplifying them is to reduce the number of mode dependent languages the user has to be familiar with. As a first step towards this end we investigate the feasibility of unified command/programming/debugging languages and the concepts on which such languages have to be based. The unification process is accomplished in two phases. First, a unified command/programming framework is defined and, secondly, this framework is extended by adding an integrated debugging capability to it. Strict rules are laid down by which to judge language concepts presenting themselves as candidates for inclusion in the framework during each phase. On the basis of these rules many of the language design questions that have hitherto been resolved this way or that depending on the taste of the designer, lose their vagueness and can be decided in an unambiguous manner.

KEY WORDS & PHRASES: Programming Environments, Monolingual Systems, Language Integration, Language Design, Command Languages, Programming Languages, Debugging Languages, Event Associations, Side-effect Recovery

†This paper is not for review; it is intended for publication elsewhere.

Rien n'est plus fécond, tous les mathématiciens le savent, que ces obscures analogies, ces troubles reflets d'une théorie à une autre, ces furtives caresses, ces brouilleries inexplicables; rien aussi ne donne plus de plaisir au chercheur.

André Weil

1. INTRODUCTION

1.1. General

A programmer interacting with a typical computer system has to be something of a polyglot. In addition to the language he is programming in, he has to be fluent in the system command language and the language of the symbolic debugger. Furthermore, various other system utilities like the text editor and the linkage editor each have their own command language, bringing the total number of languages he has to master to at least four.

This hodgepodge of languages makes fast and efficient interaction with the system difficult. There are several reasons for this. The first and most obvious one is that the user has to remember so many different details. This would be acceptable if the domains of discourse corresponding to the various interactive modes were sufficiently distinct. The point is that, at least for some modes, the opposite is true. There are profound analogies between command mode, programming mode and symbolic debugging mode, but in most existing systems a substantial intellectual effort is required to see them, because they tend to be obscured by the differences between the various languages.

Secondly, because of the heterogeneous character of the system, the user is confronted with serious interfacing problems. An especially tricky boundary to cross is the one between his program and the file system. The mismatch between the datatypes supplied by the file system and the datatypes available in the programming language force him to resort to explicit input/output operations and intricate data conversions for even the simplest of operations on files. This whole area is a source of confusion and programming errors.

These problems have not gone unnoticed. Perhaps Shaw, one of the designers of the JOHNNIAC Open-Shop System (an early time-sharing system which became operational at The RAND Corporation in January 1964), already had an inkling of the chaos that was to ensue from the separate development of command and programming languages when he wrote: *"A striking feature of the system is that the user commands JOSS directly in the same language that he uses to define procedures for JOSS to carry out indirectly"* [SHA64]. And in 1966, at the occasion of the decommissioning of JOHNNIAC, Ware, who had been closely involved with it, said somewhat optimistically: *"Those who know JOSS and perceive the friendliness of its help and reaction feel strongly that systems such as it will be one of the prominent, if not exclusive, ways of computing for the future"* [GRU79]. Reading all this fifteen years later one cannot help but get the impression that somehow the evolution of programming environments has lagged. This is not to say that no developments have taken place in this field since JOSS first made its appearance. Powerful systems like APL/700 [BUR74], the CDL2 'laboratory' [BAY80], INTERLISP [TEI78], a recent LISP environment developed at IBM [ALB79], PATHCAL [WIL80], and SMALLTALK [GKA76, ING78, BYT81], each of which in its own way provides the user with a highly integrated interactive environment, are keeping the spirit of JOSS alive. Nevertheless, the main trend has always been to let the various interactive modes influence each other as little as possible.

One reason current time-sharing systems are suffering from a lack of homogeneity is that they have to support a multitude of different programming languages. It seems sensible, in terms of implementation effort required, to provide a single environment for all languages the system is intended to support. Such an environment has of necessity to be a compromise, however, being less than perfectly adapted to each individual language. Also, language specifications almost invariably assume 'external' data (files) to have entirely different characteristics from 'internal' data (i.e. data that are local to the program). This split propagates through the whole system and cannot be hidden from the user.

Yet another problem is that most programming languages (except APL [FIV73], LISP [ALL78] and SNOBOL4 [GPP71]) do not permit the dynamic creation and subsequent modification of procedures. This essential mechanism without which a system cannot change or grow (except by adding another level of interpretation) therefore has to be supplied by the command language.

By pure coincidence there are currently two factors working in favor of a more integrated approach to system design. First, time-sharing is rapidly losing ground to personal computing and many personal systems do not have to support more than a single programming language. Secondly, system command languages have reached a point in their evolution where their similarity to regular programming languages has become so obvious that a kind of attractive force striving for even greater similarity has started to build up.

The reader who wishes to gain a broader perspective on the various issues involved should consult the paper by Sandewall [SAN78], the proceedings of the 1980 Symposium on Software Engineering Environments [HUN80], and the compilation of recent papers on programming environments by Buxton [BUX80].

1.2. Scope of this paper

For good reasons most integrated programming environments developed so far are based on languages not specifically designed to be used that way. By using an existing language the designer avoids the quicksand of shifting language specifications and the promotion effort needed to convince prospective users of the merits of his new proposal. Although these are great practical advantages, the unfortunate consequence is that the influence of integrated environments on language design remains largely unexplored. SMALLTALK [GKA76, ING78, BYT81] is an important exception.

In this article we shall look at programming environments from a language designer's perspective. Suppose a language is to be embedded in an integrated programming environment. How would this requirement affect its design? We shall attack this problem by investigating the feasibility of *monolingual* systems *in which the command language, the programming language and the language of the symbolic debugger are identical.* The three main questions to be answered are:

(I) Are unified command/programming/debugging languages feasible and on what concepts would they have to be based?

(II) Would such languages be significantly easier to use than the typical conventional user interface?

(III) In what respects would the implementation of such languages differ from the typical conventional operating system and language processor?

Before going on to a detailed discussion of basic concepts we shall first give a brief survey of the present status of command languages in §2. §3 will be devoted to a discussion of (I). In essence, what we shall attempt to do is to derive a coherent set of language concepts starting from the single requirement that it must constitute a suitable basis for a monolingual environment in the above sense. This will take up the major part of the paper. The final section will be devoted to a brief discussion of (II) and to a general evaluation of the concepts developed in §3. We shall pay no attention to (III) in this paper.

We are using the adjective *monolingual* in a somewhat *ad hoc* manner. There are, for instance, no obvious reasons to refrain from attempting to integrate the text edit mode in addition to the three modes discussed in this paper. We have not yet tried this simply because we had to stop somewhere, but attempts in this direction will almost certainly be worth-while.

Two other subjects we shall not concern ourselves with are concurrency and protection. Again, the reason is that we had to restrict the scope of our investigation in some reasonable way. The design of debugging facilities for languages allowing the manipulation of concurrent processes is a difficult task and their integration into a larger whole is probably even more difficult. As for protection, the main problem is to reconcile protection and debugging facilities. By their very nature, these tend to be in conflict with each other. For a monolingual system, a language based protection scheme allowing user defined access control would be an obvious choice, but this does not seem to make things any easier. We shall pay no further attention to the problems involved.

Needless to say, the monolingual approach as discussed in this article is not an end in itself. Although there is much to be gained by exploiting the similarities between different modes, integration of languages with dissimilar domains of discourse can only be achieved within the framework of an *extensible* base language allowing the definition of different dialects or sublanguages.

2. PRESENT STATUS OF COMMAND LANGUAGES

2.1. General

One of the basic functions of the command level is to enable the user to create, modify, and execute programs and procedures. Furthermore, it allows the manipulation of processes and large collections of more or less permanent data (files). To a first approximation existing command languages may be viewed as ordinary programming languages with powerful primitives that operate on files. At the present time command languages and programming languages are rapidly converging towards each other and any attempt to confine them to different categories would be a step in the wrong direction. The differences between them do not in any way have a fundamental character but are rather a result of the separate evolutionary paths they have followed. In order to bridge the gap that still separates them, it is useful to look at existing command languages from a programmer's perspective. The following general remarks have been inspired by three representative current command languages, namely the so-called UNIX *shell* [BOU79], IBM's TSO Command Language [IBM78], and Burroughs' Work Flow Language (WFL) [BUR77]. In §2.2 we discuss the UNIX shell in more detail and in §2.3 we draw some conclusions.

Both the shell and TSO are incrementally interpreted, that is, each command is executed immediately after it has been read by the command interpreter. WFL is compiled, reflecting the fact that it was originally intended to be used exclusively in batch mode. Neither the shell nor TSO perform an overall syntax check of command procedures that are submitted in their entirety instead of incrementally. This difference in implementation may be one of the reasons WFL is in some respects more like a conventional programming language than the other two. Among the features of WFL are declarations and a conventional evaluation mechanism. The shell and TSO, on the other hand, do not have declarations (these would be bothersome to the interactive user) and use an evaluation mechanism based on macro substitution. This is a direct descendant of the simple substitution mechanism that was among the earliest facilities introduced at the command level to make life easier for the user by allowing him to abbreviate frequently used commands.

Computation at the command level is to a large extent string oriented and consists of the synthesis of parts of command language statements to be used later on. These invoke programs (which may themselves be written in the command language), specify file parameters, manipulate the file directory, etc. It is also possible to create file names dynamically and to perform operations on existing file names. Because of this, file names have more inherent meaning than names of variables in ordinary programs. The reason for this difference is that file names are permanent entities in the system, while names of variables have a more temporary character and are usually eliminated from the executable code to gain speed. To simplify string handling all three languages have operations like concatenation, substring selection and pattern matching (the latter mainly in the shell). The availability of variable length strings and string operations, however *ad hoc* they may be, contributes substantially to the popularity of command languages as programming tools. Of course, macros thrive in this environment although macro languages are notoriously difficult to understand. The shell and TSO are no exception in this respect.

It should be stressed that the variables offered by the three languages under discussion are used as temporary storage by command procedures and are distinct from files. The split between file types and the types of local variables of command procedures is analogous to (and just as undesirable as) the split between file types and the types of local variables in programs written in ordinary programming languages (see §1.1). In addition to the type differences between files and local variables, the permanent environment as defined by the file directory has a much more involved structure than the local environment. As a result, file names have a more complex syntax than local names. Also, the creation, maintenance and inspection of the file directory requires a large number of primitives that do not have 'local' equivalents.

2.2. The UNIX shell

In this section we shall concentrate on the UNIX Version 7 shell [BOU79], both because we are most familiar with it and because it is a powerful tool offered as part of an increasingly popular programming environment. We shall first give a list of mechanisms which shell procedures can use to communicate with each other. It will serve as a yardstick for measuring the consistency and power of the concepts to be introduced in §3. It will be followed by three sample shell procedures to give the reader a concrete feeling for the two chief characteristics of command languages: their power and their chaotic character. We shall also take the opportunity to translate some of the shell concepts occurring in the examples into more familiar and/or more consistent terms.

Shell procedures can communicate through

- ☐ *Parameters:* Shell procedures may have a variable number of parameters; both keyword and positional parameters are allowed.
- ☐ *Return value:* Essentially a boolean value which is used to drive shell control structures like the **if** and **while** statements.
- ☐ *Exported variables:* In the shell a command procedure corresponds to a separate process. An entirely new environment is created each time a command procedure is activated. The environment of the caller is not accessible to the callee, except for variables that have been explicitly designated as *exportable.*
- ☐ *Shared files:* Files are essentially homogeneous character/byte strings. UNIX does not have other file types.
- ☐ *Command substitution:* The string value produced by the callee on its output port is substituted for the call.
- ☐ *Pipes:* The string value produced by one command process on its output port is sent to the input port of another command process. The latter need not wait till the former has produced its entire output, but can start processing as soon as part of it is available (depending on the kind of computation involved, of course). The pipe mechanism takes care of synchronization between the two processes.

(For ease of reference and to enhance readability line numbers have been added to the following three sample shell procedures, while keywords are **bold** face. Neither of these are shell conventions.)

Example 1

```
1  for name in ‘ls‘
2      do
3          if test −d $name
4              then echo $name′ directory - not copied′
5              else cp $name backup
6          fi
7      done
```

This is a small run-of-the-mill shell procedure. It copies all files in the current file directory to the directory *backup*. It uses one local variable (*name*) and four procedures (*ls*, *test*, *echo* and *cp*). Whether the latter are themselves written in shell language or in another language is immaterial. Procedure *ls* produces a listing of the file names in the current file directory on its output port (line 1). The *command substitution* mechanism denoted by the opening quotes in line 1 redirects this output to the caller which, in this case, is the **for** statement. The effect is, that the controlled variable *name* successively runs through all names in the current directory. The body of the loop (lines 3-6) first tests whether the current name refers to a directory by calling procedure *test* with parameters $-d$ and the value (expansion) of *name* denoted by $*name*. If the current name refers to a directory, an appropriate message is issued (line 4). The value of the parameter of *echo* in line 4 is the value of *name* concatenated with the text between string quotes. Alternatively, if the current value of *name* refers to a file, the file is copied to *backup* by *cp* (line 5).

Example 2

```
1   echo abc >x
2   y=‘cat x‘
```

This procedure illustrates the asymmetry between files and local variables in the shell. In line 1 procedure *echo* copies the value of its argument to its output port. In this case the latter is redirected to file *x* (denoted by >*x*) and the value of the argument is *abc*. If *x* already exists, its old value is replaced by *abc*; otherwise *x* is created and set to *abc*. In line 2 *cat* copies the value of the file denoted by its first argument to its output port. The value of the latter is taken as the value to be assigned to *y* by *command substitution* (denoted by opening quotes). In programming language terminology one would say that this shell procedure assigns the string value *abc* to a permanent variable and subsequently assigns the value of the permanent variable to a local variable. If *x* were a local variable and *y* a permanent one, the procedure would look quite different:

```
1   x=abc
2   echo $x >y
```

Example 3

```
1   compname=$1$2
2   echo $1′ “′$2′ $1“′ >$compname
3   chmod +x $compname
```

This example is somewhat more involved than the previous ones. The reader who is familiar with LISP may wish to take a look at its LISP equivalent first:

```
1   (lambda (f1 f2)
2           (set (concat f1 f2)
3               (list (quote lambda) (list (quote par))
4                   (list f1 (list f2 (quote par)))
5               )
6           )
7   )
```

When called with actual parameters (*quote f*) and (*quote g*) it assigns

(lambda (par) (f (g par)))

to variable *fg*. (The function *concat* in line 2 is non-standard. It is the concatenation operator for atoms.) Variable *fg* thus becomes a function variable. Its value is the composition of *f* and *g*.

Similarly, the above shell procedure is the functional composition operator for shell procedures that obey certain argument conventions (to be specified). Its arguments are the names of the two shell procedures to be composed. Its result is the shell procedure which is their composition. By convention the two arguments are numbered 1 and 2. (Remember that shell procedures can have a variable number of parameters.) In line 1 the name of the result is synthesized. It is the concatenation of the names of the two procedures to be composed (denoted by $1$2). The output port of *echo* in line 2 is redirected to a file with this name (denoted by >$*compname*). The argument of *echo* looks rather forbidding. It is best understood by looking at its value when actual values are substituted for procedure arguments 1 and 2: if the procedure is called with parameters *f* and *g* the argument of *echo* evaluates to

```
f ‘g $1‘
```

Procedure *echo* writes this value to file *fg*, which is made executable by a call to procedure *chmod* in line 3, i.e. *fg* becomes a shell procedure itself.

What happens when *fg* is executed? Both *f* and *g* are supposed to have one argument and to produce their result on their output port. The same argument convention should apply to *fg*. By using *command substitution* (denoted by opening quotes) the output port of *g* is redirected to the procedure *fg* itself. Compare this with examples 1 and 2. In this way the result of *g* becomes the argument of *f*.

2.3. Conclusions

In view of the foregoing it will come as no surprise that command languages and programming languages have evolved into competing tools. Many programs consist of a main program written in a command language and one or more procedures written in a programming language. (For historical reasons the latter are often considered to be the ‘real’ programs.) Although such a ‘mixed mode’ implementation may relieve the programmer from a lot of work because many things are easier to program in a command language than in a regular programming language, the resulting programs tend to be difficult to understand and highly non-portable. This is partly because most command languages contain all kinds of strange features and suffer from a lack of proper syntax. For another part it is caused by the highly irregular interface between both types of languages and the fact that standardized command languages do not exist. The universal lack of adequate command level debugging facilities does not improve things either.

Clearly, programming languages are too weak to fend off the intrusion of command languages. The existence of all kinds of powerful but unstandardized command languages poses a serious threat to the effectiveness of any language standardization effort. Straightforward standardization of command languages in their present form (whatever that may mean) would only consolidate an already unsatisfactory state of affairs and would therefore be undesirable. In the sequel we hope to show that the current competition between command languages and programming languages is but a prelude to the emergence of more powerful languages encompassing both.

3. BASIC CONCEPTS FOR A MONOLINGUAL PROGRAMMING ENVIRONMENT

3.1. General principles

Our proposal for finding a suitable conceptual basis for a unified command/programming/-debugging language may easily fall short of its goal if proper guiding principles are lacking. We need some simple rules to protect us from the enormous amount of features offered by current languages and systems and to aid us in finding our way towards a coherent whole. Fortunately, such rules are inherent in the concept of integration itself:

(A) A linguistic concept is eligible for inclusion in the set of basic linguistic concepts only if it adds substantial power *in all three modes.*

(B) The semantics of each concept must be *mode independent.*

(C) On the basis of the resulting set of linguistic concepts it must be possible to provide the user with adequate facilities in all three modes.

We use the word *mode* in the sense of *functional setting* or *kind of activity*. In this sense it denotes something more immediately pertaining to the kind of activity the user feels he is involved in than to something specific in the system itself.

The above requirements are not as vague as they may seem at first glance. A vague term like *adequate facilities* can be given a more precise meaning by using the facilities offered by existing integrated as well as non-integrated environments as a yardstick. In fact, (A), (B) and (C) are so restrictive that they are occasionally in conflict. Nevertheless, they have proved to be very useful guidelines and we shall stick to them as closely as possible.

Requirement (A) forces us to look at each concept from three different viewpoints. In most cases, at least one of these provides an unexpected perspective, even if it turns out that the concept in question is tied too closely to a specific mode to be of general use. Occasionally, the opposite happens and a concept turns out to have unsuspected applications in modes that initially may have seemed foreign to it.

In the next section we first develop the concepts underlying a unified command/programming language. We then add an integrated debugging capability to it in §3.3. Requirement (A) guarantees that the resulting unified command/programming/debugging framework is largely independent of the order in which concepts are added.

3.2. Integration of command and programming language

3.2.1. Information and control flow

In conventional programming languages the three chief mechanisms for communication between program units are:

- ☐ Procedure parameters.
- ☐ Procedure return value.
- ☐ Global variables.

Their command level equivalents are (§2):

- ☐ Program and command procedure parameters.
- ☐ Program and command procedure return value.
- ☐ Files and *exported variables.*

Unification of both sets of mechanisms is achieved as follows:

- ☐ The distinction between programs and procedures is eliminated.
- ☐ The distinction with respect to type and naming between files and variables is eliminated.

We shall elaborate on both points. First, in a unified framework calling a procedure is indistinguishable from running a program. The command level corresponds to interactive programming at the highest procedural level of the system. To preserve symmetry between this level and deeper procedural levels it is necessary to introduce an **interact** construct, allowing interactive mode to be entered at the point at which it occurs. The command level is thus an implicit **interact** at the highest procedural level, but interactive programming mode may be entered at deeper levels as well. Execution of an **interact** does *not* cause an environment switch. The **interact** construct is similar to the LISP **eval** function. Its application in interactive debugging mode is discussed in §3.3.2.

The semantics of shell concepts like *command substitution* and *pipes* (§2.2) can for the most part be expressed in terms of ordinary procedures. There is one exception, however. Producing and consuming processes which communicate by means of a pipe-like method cannot always be modeled by function composition. If the producer happens to create an infinite output stream, the pipe mechanism allows the consumer to start doing useful work on a time-multiplexed basis as soon as part of its input is available. If the pipe is modeled in terms of function composition, each functional component has to finish before the next component can start, so in this case the model is inadequate. The problem can be solved either by introducing special language features, like interprocess communication or co-routines, or by using some form of lazy evaluation. As this goes at the expense of a relatively large increase in complexity of the language, we do not think it worth-while to explore these possibilities any further in the present paper.

In a unified framework, the analogues of files are *permanent variables*. (§2.2, example 2). We shall call all other variables *local*. Conventional global variables are thus also called local in our terminology. The first step in unifying permanent and local variables is to abolish the distinction between permanent and local *data types*. For instance, suppose we are maintaining an on-line telephone directory. In a system in which the command and programming language are different, the following three steps are required to modify a directory entry:

(1) Enter a special purpose 'telephone directory editor' (or the standard text editor if the directory resides on an ordinary text file).

(2) Modify the entry.

(3) Return to command mode.

On the other hand, if we are maintaining the directory in a system with a unified command/-programming language, the information can be represented using a suitable data type, such as an associative table that maps names on telephone numbers. To modify a table entry a simple command mode assignment suffices:

```
telephonenumbers[person] := newnumber;
```

No special program or mode switch is needed to modify the information.

From now on we shall be using a single type system throughout. Variables can differ with respect to their lifetime, but this does not imply any difference with respect to the types of values they can have. Type definition and type checking in a unified command/programming language will be discussed in the next subsection.

The second step in eliminating the differences between permanent and local variables is to unify the *naming* mechanisms at the two levels, i.e. to give the permanent and local environment the same basic structure. This is the subject of §3.2.3.

Integration of conventional control flow mechanisms, like **if**, **for** and **while** statements, does not present any serious problems. They are already present in both kinds of languages. Some desirable features of command procedures, such as multiple return values, failure signals, keyword parameters, etc., are seldom found in programming languages, but their inclusion in a unified framework is a relatively minor issue.

Some command and programming languages allow a modest form of *exception handling*. This important subject will come up in a natural way in the context of the integration of debugging tools (§3.3.4.1).

3.2.2. Type definition and type checking

The user of a unified command/programming language must be able to handle both small scale data, like integers and short strings, as well as large scale data, like entire texts, arrays of numbers, trees and directories, with equal ease. In view of this it is appropriate to introduce an *abstract type definition* facility in the language. In conjunction with sufficiently powerful basic types (including dynamic arrays and associative tables) it should enable him to define most required data types himself. Directories are somewhat special and are discussed in the next subsection.

A unified command/programming language has to be *complete* in the sense that procedure and type declarations are not supplied by an outside source, but must be created and manipulated in the language itself. This, and the fact that statements in the language are entered both incrementally and in the form of predeclared units, makes it necessary to consider type systems from a broader viewpoint. In the following we shall briefly discuss three aspects of them, namely:

☐ *Elastic* type checking.

☐ The need to introduce procedure and type valued variables.

☐ The influence of modifications on type consistency.

Insofar as our account is incomplete, the reader is referred to an interesting recent article by Goodwin [GOO81].

A central issue with type systems is the *moment* at which the rules are checked. Procedure and type declarations can be checked as a whole, but interactive commands have to be checked incrementally. It is not clear how both situations can be captured by a single, static type system. An analysis of the moment at which the checks *could* be performed may shed some light on the problem. A typical procedure will go repeatedly through some or all of the following stages:

(1) Create or modify procedure declaration.

(2) Compile.

(3) Include in library.

(4) Combine with referenced procedures from a library.

(5) Call procedure.

(6) Execute body.

Not all type checks can be performed during stage 1, because not all information is statically available. An example is the check on the type of external procedures. This check cannot be performed before stage 4 above, but could be postponed to stages 5 or 6. In a typical statically typed programming language most checks are performed during stages 1 through 4. It is interesting to note that operations on permanent data (files) are almost always checked during stage 6. If a file name is created dynamically no earlier check is possible, but statically known files could be checked earlier. This is seldom done, however. Also, checking the type of external procedures in stage 4 is often done rather cursorily or not at all. In a typical command language all checks are postponed to stages 5 and 6.

For a unified command/programming language the simplest solution is to perform all type checking at run-time. In this scheme there is no distinction between incremental execution and the execution of predeclared units with respect to the moment at which type checking is performed.

Another - much better - method is to check type consistency at the earliest possible moment. This leads to an *elastic* type system covering the whole range from strictly static typing to completely dynamic typing. In such a system the type rules are checked *as soon as sufficient information is available.* This amounts to static typing when full static information is available, and to dynamic typing when no static information is available at all. Furthermore, cases in which there is only partial static information can also be handled.

The *completeness* property mentioned in the first paragraph of this section has important consequences. The monolingual equivalent of the creation of a new program is the declaration of a new procedure at the command level. When completed, the declaration becomes an object of type *procedure* and is given a name, i.e. it becomes the value of a procedure variable. Depending on the scope of the name given to it, it may either become a local procedure or a permanent one. In the former case it is destroyed on return from the command level, i.e. on log-out.

As the command level corresponds to incremental programming at the highest procedural level of the system, all facilities offered by it are shared by other procedures. As a consequence a unified command/programming language must necessarily allow *nested procedure declarations, procedure variables, and procedure parameters.* Similar considerations lead to the introduction of *type valued variables.* The value of an object of type *procedure* or *type* is the original declaration used in its creation. But, hidden from the user, it may contain an optimized (compiled) version of the declaration which is used instead of the original source text when the procedure or type is invoked.

For editing purposes it is necessary to decide which program units are *modifiable*†. Because there is a natural conversion from an object of type *procedure* or *type* to an object of type *string* and vice versa, procedure or type declarations can be modified in a straightforward manner. Modification of a single statement, however, can be achieved only *indirectly* by looking for the procedure or type declaration to which it belongs and by assigning a completely new declaration (which only differs from the previous one in the modified statement) to the original procedure or type variable.

It is sufficient to have only a few high level constructs that are modifiable provided that all constructs in the language are covered either directly or indirectly. This may lead to the introduction of additional types, if procedures and type definitions are not the only modifiable units. If a program modifies one or more of its own modifiable units, it is *self-modifying.* The possibility of self-modification is thus a consequence of *completeness.*

Modifications may easily lead to type inconsistencies. For instance, if a type definition is modified all instances of the old version may become incompatible. Similarly, replacement of a procedure may lead to type inconsistencies between the new version and existing program units referring to it. An elastic type system, whose purpose it is to report type inconsistencies at the earliest possible moment, would need extensive backward linking in order to be able to perform its duty in these cases. Although not easy to implement, this kind of service from a type system is quite unheard-of in conventional systems.

†We shall not pursue the integration of editing primitives here, but note in passing that the string manipulation component of the language may be viewed as the forerunner of an integrated editing facility. Another interesting point is, that the positioning operations needed for editing source text can also be used to identify control flow events (breakpoints) in debugging mode. See §3.3.

3.2.3. Environments, directories and abstract type definitions

Both in command and programming languages there is a bewildering variety of mechanisms to associate names with values. Furthermore, unlike the set of local names in a conventional program, the set of names at the command level has a dynamic character. These large differences between the permanent and local environment (see also §2.1) are not acceptable in a unified command/-programming language. In this section we show that, on closer inspection, most of the mechanisms involved turn out to be rather similar and can be unified by allowing the user to structure both the local and permanent environment himself. An immediate consequence is, that variables can be created and destroyed dynamically irrespective of their scope.

Let an *environment* be a linear list of (name,value) pairs. Environments will be denoted by

$$\{(N_1,V_1), \ldots, (N_n,V_n)\}.$$

When the value of name *N* in environment *E* is needed (this will be denoted by *E.N*), *E* is searched from left to right for a pair with name part *N*. The corresponding value part is the value of *E.N*. A value *V* can be associated with name *N* in environment *E* by searching *E* from left to right for a pair with name part *N* and by substituting *V* for the corresponding value part. Different pairs in *E* can have the same name part, but only the leftmost one is affected. If *N* does not occur in *E*, the new pair (*N*,*V*) is appended at the right-hand side of *E*. The *join* of two environments *E* and *F* is obtained by appending the pairs of *F* at the right-hand side of *E* in the same order in which they occur in *F*.

An *object directory* (the successor of the conventional file directory) is an environment that maps object names on objects. An object directory *D* containing objects *A*, *B* and *C* can be described by

{(A:objectA), (B:objectB), (C:objectC)}.

Obviously, one of the names in the directory could itself be a directory (i.e. a sub-environment). In this way, tree-structured and even more general directory systems can be described. Supposing *B* in the above example is a directory containing objects *B1* and *B2*, the resulting structure can be described by

{(A:objectA), (B:{(B1:objectB1), (B2:objectB2)}), (C:objectC)}.

Selection of object *B1* from directory *D* is then achieved by *D.B.B1*.

In the shell so-called *path names* are used to specify the position of a file in the directory tree with respect to the 'current directory' or the root of the tree. The above selection is similar both to such path names as well as to (repeated) field selection from instances of abstract types. The shell user can also specify a *search path*, i.e. a linear list of directories in which the binding (interpretation) of each command read by the shell is looked up. For instance, suppose the user wants to execute command *P*. The first executable file with name *P* encountered when searching the given list of directories is then taken to be the program or command procedure to be executed. The interpretation of *P* can be changed not only by replacing *P* itself, but also by placing another version with the same name in a directory which is closer to the start of the search path. Similarly, many systems allow the definition of search paths consisting of linear lists of procedure libraries for use by the linkage editor. If the first library does not contain the procedure to be linked, the linkage editor goes on to the second library, etc. A search path can simply be described as the *join* of a series of environments.

Now that we have brought out these similarities, the next question is: how can environments be incorporated in a unified command/programming language? Rather then incorporating environments as such, a better solution is to improve the environment-like properties of instances of abstract data types. An example may clarify this. The PASCAL **with** statement ([JWI75], §7) can be looked upon as installing the record variable mentioned in its header as a component of the environment. Given the declarations

```
type date=
    record day: 1 .. 31;
           month: 1 .. 12;
           year: integer
    end;
var  mybirth, today : date;
var  myage : integer;
```

and appropriate initializations, one can write

```
print(today.day);
print(today.month);
print(today.year);
myage := today.year − mybirth.year;
```

PASCAL allows the above statements to be abbreviated to

```
with today do
    begin
          print(day);
          print(month);
          print(year);
          myage := year − mybirth.year;
    end;
```

i.e. within the scope of the above **with** statement *today.day* may be abbreviated to *day*, etc.

The PASCAL **with** statement can be generalized by permitting abstractly typed objects with procedural components and *own* variables instead of just records with passive fields. Such generalized versions of the **with** statement are the CDL2 **focus** statement [BAY80], and the **scan** expression in SUMMER [KLI80]. The **use** clause in ADA [DOD80] serves a similar purpose.

We are not finished yet, because there is still an essential difference between environments and ordinary abstractly typed objects. An instance of an abstract type has a fixed number of fields, while the number of elements in an environment or directory may vary dynamically. This problem can be solved by allowing a 'table of contents' to be added to an abstract data type and by adapting the rules for the interpretation of names in generalized **with** statements as follows: the occurrence of name x within the scope of a generalized **with** statement with argument *OBJ*, is equivalent to

```
OBJ.x
```

if the type of *OBJ* statically includes a field x, and to

```
OBJ.contents['x']
```

if it does not. In the latter case, the table of contents of *OBJ* is accessed via its *contents* field with key x. String quotes have been added to indicate that the table key is a literal. In the first case it is assumed that fields are not evaluated, so no quotes are necessary.

Execution of a generalized **with** is rather similar to switching to a different 'current directory' in command mode (*cd* command of the shell). The equivalent of a search path (see above) can be obtained by nesting generalized **with** statements or by allowing a list of objects instead of only a single object to be specified in the header of a **with**. Again compare this with the PASCAL **with** statement which may be applied to a single record variable as well as to a list of record variables.

3.3. Integration of debugging facilities

3.3.1. General

Grishman has argued the unification of programming and debugging languages as follows: *"I would like to emphasize particularly that the debugging language should be similar to the source language. Since the data structures involved are those of the source language, it is only natural to use statements from the source language in debugging, rather than to try to put together a new language which reflects the data structures of many languages. Also, the resistance of most users toward learning a new language in order to debug their programs is not to be underestimated; a language with which they are already partly familiar helps to overcome this resistance"* [GRI71]. Grishman does not advocate the total integration of programming and debugging languages, but leaves room for special facilities that are only available in debugging mode.

The designers of SNOBOL4 have gone one step further in an attempt to integrate the two kinds of language completely. The tracing facilities of the original version of SNOBOL4 as defined in [GPP71], chapter 8, constitute a more or less separate set of functions not too well integrated into the rest of the language. To remedy this defect Hanson has defined an interesting language extension [HAN76, HAN78]. It consists of a mechanism called *event association* allowing the programmer to associate a SNOBOL4 procedure with assignments to variables, procedure calls, etc. Hanson has also investigated the application of event associations for purposes other than debugging.

In a monolingual system interactive debugging is indistinguishable from incremental programming. Likewise, statements that have been added to a program for purposes of debugging cannot be distinguished from 'ordinary' statements. Because command mode corresponds to incremental programming at the highest procedural level, interactive debugging at that level is indistinguishable from command mode itself. Although the user may have a very specific idea regarding the mode he is operating in, this is immaterial as far as the system is concerned. As was pointed out in §2.3, conventional systems do not only suffer from a chaotic command language but also from a lack of command level debugging facilities. Obviously, the latter cannot happen in a monolingual environment.

Implicit in the above is the availability of an **interact** primitive allowing the interactive insertion of statements at the point at which it occurs. This construct was already introduced in §3.2.1 as part of the development of a unified command/programming framework. In addition to an **interact**, the user needs two other tools to aid him in debugging:

(1) He must be able to associate actions with the occurrence of certain computational *events*. He may for instance wish to switch to interactive mode whenever the value of a certain variable becomes zero during the execution of a certain procedure or to display the values of the actual parameters when a certain procedure is called. Furthermore, in order to catch an event *the user should not be forced to locate all points where it can occur*. To this end we shall introduce an event association mechanism similar to that of Hanson. Although our event associations will obey somewhat different rules than his we shall use the same name. Event associations will be discussed in §3.3.2.

(2) The second tool needed for debugging is *side-effect recovery*. For instance, the user who wishes to inspect the top element of a stack may be forced to use the stack's pop operation for lack of an accessible top-of-stack field. Although in this particular instance he could probably undo the side-effect of the pop operation rather easily by pushing the popped item back on the stack, in other cases this might be difficult or even impossible. The inverse operation(s) required to restore the original state may be complex and time-consuming or, perhaps even worse, the user may be unaware of the precise side-effects some of his actions are causing. What is needed is a general

mechanism to evaluate expressions and undo all their side-effects (except, of course, the delivery of the result). This will be the subject of §3.3.3.

As a result of the integration of debugging facilities all unified command/programming constructs introduced in the previous section become available in debugging mode, while event associations and side-effect recovery become available as general programming tools and in command mode. Requirement (A) of §3.1 stipulates that all three modes should benefit from this. The merits of normal programming constructs in debugging mode were set in evidence by the quotation at the beginning of this section. The applications of event associations and side-effect recovery in programming and command mode are perhaps less evident and will be discussed in §3.3.4.

We conclude this chapter with a discussion of the disadvantages of integrated debugging in §3.5.

3.3.2. Event associations as a debugging tool

Integrated debugging facilities should not force the user to plan his debugging activities beforehand or to make temporary modifications to the source text of the program units involved. Event associations should therefore be designed in such a way that the user can control and monitor execution interactively by jumping from one event to another without modification of any source text whatever. Furthermore, in order to enable the user to inspect variables in the current environment with the aid of ordinary language constructs, triggering of an event association should *not* cause an environment switch. Although these two requirements are important, they are far from sufficient to guarantee the proper behavior of event associations. A very strict discipline has to be imposed on them to ensure their predictable behavior and their smooth interaction with other language constructs. The set of requirements given below is in no sense complete or definitive, but is meant to give an indication of the many issues involved.

To simplify the following discussion we first introduce a notation for event associations:

<event association> ::= [<label>] **when** <event> **do** <action> **od**

Some examples are:

Example 1

```
when x is modified
     do
        interact
     od;
```

This event association causes interactive programming mode to be entered when *x* is modified.

Example 2

```
when P is called
     do
        display x1, ..., xn
     od;
```

The values of the parameters of *P* are displayed just before execution of the body of *P* starts.

Basically, when an <event association> is executed as part of the normal flow of execution, the corresponding (<event>,<action>) pair is *connected.* It simultaneously gets *armed.* In the armed state the <action> is *triggered* when the specified <event> occurs. For reasons to be explained below, an event association can also become *temporarily disarmed.* In that state it remains connected, but it cannot be triggered. If conditions are favorable again, it is *re-armed.* When it is no longer needed, an event association is *disconnected.* This either happens automatically or can be done explicitly by means of a *disconnect* statement containing a reference to the <label> of the <event association> to be removed.

To be useful as a debugging tool event associations have to meet at least the following requirements:

(1) Side-effects due to the testing of the event part of an event association are automatically undone after the test has been performed, independently of whether the test succeeds or fails. In this way the behavior of event associations is guaranteed to be independent of the number of actual tests performed by the implementation. See example 3 below.

(2) When an event association is triggered, its action part is interpreted in the current environment, i.e. activation of the action part does not cause an environment switch. Apart from unwanted side-effects, this convention allows straightforward inspection and modification of variables in the trigger environment with the aid of ordinary language constructs in the action part. Side-effect free inspection requires special constructs and is discussed in §3.3.3.

(3) The event part is statically bound to the environment in which the **when** occurs (although this may of course depend on the general binding strategy used by the system). Binding of the action part is dependent on the type of event specified. Normally, it will be bound to the same environment as the event part, but if the event involves an environment switch, the action part will be dynamically bound to the new environment. If one or more of the variables occurring in the event part of an event association become temporarily inaccessible due to an environment switch, the event association is *temporarily disarmed.* If one or more of these variables are destroyed, because the environment to which they belong is destroyed, the event association is *disconnected.* For instance, all event associations connected during execution of a procedure are disconnected on procedure exit. See example 4 below.

(4) No constraints are imposed on the statements making up the action part. In particular an action part may itself contain event associations. In keeping with (3), these stay connected as long as the environment to which their event parts are bound has not been destroyed, i.e. they are *not* automatically disconnected when the action part terminates. Event associations can thus be 'exported' from an action part. See example 4 below.

(5) No constraints are imposed on the position of event associations with respect to other statements.

(6) Potential trigger conflicts must be eliminated by imposing a linear order on the set of armed event associations. Such an order can, for instance, be based on 'seniority' or on some linear ordering of environment components. Even if resolved properly, trigger conflicts easily lead to unforeseen complications because of side-effects of the action parts involved.

We give two further examples to illustrate the various aspects and possibilities of event associations.

Example 3

```
when x < stack.pop
     do
        display x
     od;
```

At the moment *x* becomes less than the top element of *stack* (if any), the value of *x* is displayed. The event may occur either when a new value is assigned to *x* or *stack*, if *stack* is popped, or if a new value is pushed on *stack*. The stack pointer is reset to its original value by the side-effect recovery mechanism after the event test has been performed. See also example 1 of §3.3.3.

Example 4

Suppose the following event association is entered interactively at the command level:

```
when P is called
      do
         interact
      od;
```

A subsequent call

$P(x_1, \dots, x_n)$;

triggers the event association just before execution of the body of *P* starts. The **interact** statement in its action part causes a (nested) switch to interactive mode. Neither the triggering of the **when** nor the execution of the **interact** causes an environment switch, so any succeeding interactive statements are interpreted in the context of *P*. The user can now connect an event association to the current invocation of P, for instance:

```
when x = 0
      do
         interact
      od;
```

After leaving interactive mode, execution of the body of *P* starts until the value of *x* becomes equal to 0. At that moment interactive mode is re-entered, etc.

3.3.3. Side-effect recovery as a debugging tool

As was explained in §3.3.1, the primary purpose of side-effect recovery is to protect the user from unwanted side-effects when he is inspecting the state of his program in debugging mode. Any side-effects caused by testing the event part of an event association are automatically undone, independently of whether the test succeeds or not (§3.3.2), but everywhere else (and especially in the action part of event associations) side-effects have to be recovered explicitly with the aid of special constructs.

It is sufficient for the purpose at hand to introduce *recovery brackets* which transform an expression into its side-effect free equivalent. We shall use the following notation:

```
<expression> ::= ... | <probe>
<probe>      ::= probe <expression> endprobe.
```

By adding <probe> as an alternative to <expression>, the first rule ensures that a <probe> may occur at all syntactic positions where an <expression> is allowed. The second rule says that key words **probe** and **endprobe** play the role of recovery brackets. Evaluation of an expression enclosed in recovery brackets proceeds in the normal fashion, but when evaluation terminates the process is restored to the state it was in just before evaluation started, except that the new statement pointer and the result(s) of the expression are retained. Any success/fail signals returned by the expression are counted as results and are thus retained as well. If a result shares its value with a variable, a complete recursive copy of the shared object is made before the process state is restored. This copy is returned instead of the shared object itself so as to prevent the result from being affected by the recovery process. Sharing of values between results does not present a problem and is not undone by the copy process. If an expression does not have side-effects, enclosing it in recovery brackets has no effect.

As the program state comprises the permanent environment, recovery of permanent side-effects is implicit in the above definition. Permanent side-effects are the equivalent of side-effects due to file input/output in a conventional system. Automatic recovery of terminal input/output and other forms of man-machine interaction merits special attention, but is best treated in the context of conditional side-effect recovery. Both subjects will be discussed in the next subsection.

We conclude this subsection with some examples of the application of side-effect recovery in debugging mode.

Example 1

display probe stack.pop **endprobe**;

The *pop* operation returns the top element of *stack* and decrements the stack pointer. If enclosed in recovery brackets, the stack pointer is restored to its original value and the *pop* becomes a non-destructive read-top-of-stack operation.

Example 2

display probe text.nextline **endprobe**;

An object *text* consisting of lines of text is accessed sequentially through its operation *nextline*. An internal line pointer is incremented every time a line is retrieved. Putting the access between recovery brackets allows the user to look ahead in the text without disturbing the line pointer.

Example 3

display probe $P(x_1, \ldots, x_n)$ **endprobe**;

More general forms of look-ahead are also possible. For instance, suppose one would like to take an advance look at the results of the next call to a procedure *P*. Assuming the context is right, this can be done without affecting the state of the process by putting the call to *P* between recovery brackets.

3.3.4. Applications of debugging constructs in other modes

3.3.4.1. Event associations in other modes

Basically, event associations are *production rules* [DAK77]. The set of armed event associations constitutes a (dynamically varying) *production system* which is superimposed on the otherwise object oriented base layer of the language, thus lending it a hybrid aspect. Of course, event associations do not add essential computing power. They are strictly a matter of convenience, just like most other language features. Nevertheless, they are better suited to some applications than conventional constructs. In this section we show that the application range of event associations is sufficiently broad to justify their inclusion into the set of basic linguistic concepts (requirement (A), §3.1).

The language is assumed to provide facilities for the monitoring of at least the following events:

- ☐ Assignment to a given variable or field. See example 1 of §3.3.2.
- ☐ Truth or falsity of a given *boolean* expression or, if expressions return a success/fail signal in addition to their result, success or failure of an *arbitrary* expression. See example 3 of §3.3.2.

This is by no means an exhaustive list of useful events, but it is sufficient for our present purpose.

Example 1

```
procedure P(x_1, ..., x_n)
{
  ...
  if ... then flags.E := true fi;
  ...
};


when flags.E = true
     do
       ...
     od;
```

Event associations can be used as *exception handlers*. In this example procedure *P* raises a user defined exception by setting field *E* of object *flags* to **true**. Object *flags* contains boolean variables associated with various exception conditions. The value of *E* is monitored by a **when**, whose action part handles the exception. If the **when** happens to be temporarily disarmed at the time the exception is raised, handling of the exception is delayed till the **when** is re-armed.

Example 2

```
when x is modified
     do if R(x,y) fails
          then y := P(x)
        fi
     od;
when y is modified
     do if R(x,y) fails
          then x := Q(y)
        fi
     od;
```

Suppose that x and y have to satisfy an invariant one-one relation R, but that external factors can cause variations in both x and y (but not simultaneously). Assuming that $R(x,y) \Leftrightarrow y=P(x) \Leftrightarrow x=Q(y)$, R is kept *invariant* by the above pair of event associations.

One application of this is in keeping a subsystem consistent when one of its components is replaced. This can be achieved by defining a suitable event association which monitors the relationship between the various components. If the relation no longer holds, because one of the components has been modified, the action part rebuilds a consistent version of the subsystem.

Some systems include a software maintenance facility with its own special 'maintenance language' (e.g. the UNIX *make* program [FEL79]). With every subsystem a maintenance procedure is associated which must be executed every time a change is made. In a monolingual environment there is less need to introduce a separate maintenance language, not only because the compile and link (bind) phases are completely hidden from the user but also because the structure of a subsystem can be described by means of an abstract type definition, so no separate language for this purpose is needed. In addition to this, event associations permit maintenance actions to be initiated *automatically*.

Example 3

Define the semantics of a **guard** as

guard EXPR $\equiv$ **when** EXPR **fails do abort od.**

Should the evaluation of EXPR ever fail while the left-hand **when** is armed, execution of the offending statement is terminated, a warning is issued and control is returned to the command level. For example, consider

guard prime(n);

which causes an **abort** when a composite integer is assigned to *n*.

Apart from the **abort** which it may cause, a **guard** never has any side-effects. Compare also with example 2 of the next subsection.

3.3.4.2. Side-effect recovery in other modes

In addition to its usefulness for debugging, a side-effect recovery mechanism like the **probe** construct introduced in the previous subsection also has applications in algorithms involving exploratory actions (backtracking) and error recovery. The **probe** performs unconditional side-effect recovery. A slightly modified version allowing conditional recovery is also useful. It consists of a pair of *conditional recovery brackets* with the following syntax:

```
<expression> ::=  ... | <try>
<try>        ::= try <expression> endtry.
```

try EXPR **endtry** is equivalent to EXPR itself if the evaluation of EXPR succeeds. Otherwise, it is equivalent to **probe** EXPR **endprobe**. Side-effects are thus recovered only if the evaluation of the expression enclosed in conditional recovery brackets fails.

In the following example we show how the **try** construct can be applied to the parsing of non-LL(1) notions.

Example 1

Suppose a language has a **case** expression with the following syntax:

```
<case expression> ::= case <expression> of <case entries> esac
<case entry>      ::= <keys> <expression>
<keys>            ::= <key> [<keys>]
<key>             ::= <integer> :
<expression>      ::= <integer> | ...
```

For instance,

```
k := case n of
     1: 2: k+1,
     3:    0
     esac;
```

is a legal <case expression>.

Rule <keys> always produces at least one <key>. Both <key>s and <expression>s can have an <integer> as initial symbol. We assume that <expression>s do not contain colons. The following procedure uses a **try** to permit a <key> to be parsed without regard to its non-LL(1) character:

```
procedure keys()
{
  if key() fails then error( ... ) fi;
  do forever
     if try key() endtry fails
        then exitloop
     fi
  od;
  return
};
```

Can the meaning of automatic side-effect recovery be extended so as to encompass man-machine interaction in a satisfactory way? In discussing this question we shall assume that interaction takes place through a (possibly dynamically varying) number of *viewports*. If viewports are treated in the same way as other objects, modifications to a viewport requested by a **probe** or **try** construct will take effect immediately, even though this may subsequently turn out to have been premature. Consider the previous example. After having parsed the first <key>, procedure *keys* looks for additional ones. It does this by calling procedure *key* from within a **try** statement. Because it proceeds on the assumption that another <key> follows, *key* may issue an error message, only to discover afterward that the assumption was erroneous and that what it was parsing was not a <key>, but an <expression>. It then fails and all its side-effects are undone by the enclosing **try**. Among other things this means that the error message issued by <key> has to be canceled in some way, but even if initially marked as tentative its (probably rather brief) display would be highly confusing to the unsuspecting user. If it had been assigned to a permanent object instead of to a viewport, there would have been no problem because the user would never have seen it.

In order to minimize the amount of irrelevant information presented to the user, viewport modifications requested by a **probe** or **try** construct have to be postponed or even suppressed altogether. To some extent this can be achieved by queuing them internally till either a user feedback is requested or till the current **probe** or **try** terminates. In this way isolated messages that are not part of a dialogue can be discarded by the side-effect recovery mechanism before they get a chance to be displayed. Premature displays are still possible, however. In the above case, for instance, the error message is valid only if the alternative chosen turns out to be the the right one, independently of whether a request for user input follows production of the error message or not. To achieve this, either procedure *key* must be changed or, more generally, a *deferred evaluation* mechanism must be added to the language. In both cases, the fact that *key* is called from within a **try** must be known in advance and is reflected in the code.

Again consider example 1. If the source text is stored as a permanent object, the compiler can simply re-access the ambiguous part of the text each time procedure *key* fails, because the enclosing **try** restores the source text pointer to its original value. If the user interacts with the compiler directly, it is quite unreasonable to expect him to provide the same text repeatedly only because the compiler does not know how to parse it correctly the first time. In this particular case the problem can be solved by moving the request for input out of the recoverable section, but this is an *ad hoc* solution without obvious generalization.

The overall conclusion must be that recovery of user interaction cannot be performed in a straightforward way.

The **try** construct is a direct descendant of its namesake in the SUMMER programming language [KLI80]. The SUMMER **try** is more powerful in that it permits several alternatives to be tried in turn until a given condition is met. Experience shows that the simpler **try** and **probe** constructs presented here are adequate in most cases. This may depend on the type of application envisaged for the system however.

In the next two examples we show how side-effect recovery can be used in the definition of **assert** and **undo** constructs.

Example 2

In addition to the **guards** discussed in the previous subsection conventional positional assertions are also useful in all modes. They may take the form

<assertion> ::= **assert** <expression>.

If the evaluation of the expression part fails, execution of the offending statement is terminated, a warning is issued and control is returned to the command level. If the expression part succeeds, the assertion as a whole succeeds and no further action is taken. As they do not belong to the computation proper, assertions must not return any values or cause any side-effects (except the obvious one in case of failure). Enforcement of this rule cannot be left to the programmer, nor is it practical to limit the expression part of assertions to intrinsically side-effect free expressions. The alternative is to use dynamic side-effect recovery. In view of this, suitable semantics for the **assert** construct are

```
assert EXPR ≡ if probe EXPR endprobe fails
                 then abort
              fi.
```

Example 3

The **probe** and **try** constructs are useful only if the user has advance knowledge about the tentative character of the expression involved and the conditions under which recovery is to be performed. When working interactively, the user wants to be able to undo the effects of the previous statement simply by typing **undo** (or something similar) *after* it. One might say that all interactively entered statements have a tentative character, but that the conditions under which they have to be revoked are vague and impossible to specify in advance. If an **undo** is introduced, the unpredictable character of its interactive use makes it necessary to execute each interactive statement in a reversible way.

3.3.5. Disadvantages of fully integrated debugging

Integration of debugging facilities also has its drawbacks. An unavoidable consequence of making full programming power available in debugging mode is that the debugging statements themselves may contain bugs that are as serious as the ones the user is looking for.

A second problem is, that the user cannot inspect the entire program state at every point. Although he can switch to other environments with the aid of the generalized **with** statement (§3.2.4) and even inspect the *own* variables of objects in the current environment by means of **probes** and suitable event associations, some variables, such as the local variables of the caller of the current procedure, cannot be accessed in this way. Conventional symbolic debuggers circumvent this problem by allowing the user access to the implementation. In a monolingual environment this is unacceptable for at least three reasons: it jeopardizes the integrity of the language system; it forces the user to familiarize himself with the implementation; and it imposes severe constraints on the freedom of the implementor because, if implementation dependent notions are to be included in the language, the architecture of the underlying (abstract) machine has to be part of the language specification. It should be added that at least the first two points apply to other language systems as well. Implementation dependence is simply never acceptable.

An implementation independent way of resolving the conflict would be to introduce a mode dependent naming convention. It would consist of some simple rules to add extensions to names so as to make them unique. These rules would only apply in debugging mode and would enable the user to access all variables, fields, etc., irrespective of the environment prevailing at the time debugging mode was entered. Clearly, this solution violates requirement (B) of §3.1 which says that the semantics of all language constructs must be mode independent. Unfortunately, there seems to be no way to resolve the conflict if we have to stay within the bounds dictated by (B). This means we have reached the limits of what our integration method will allow us at this point.

Finally, one should keep in mind that event associations, although clearly an important concept, are not yet well understood. Additional research is needed on this point.

4. EVALUATION AND CONCLUSIONS

The main obstacle in reaching valid conclusions regarding the merits of the monolingual approach towards programming environments is, of course, the lack of an operational system. Nevertheless, it may be useful to attempt a brief evaluation of the ideas presented in the previous sections.

The basic idea is to simplify programming environments by reducing the number of mode dependent languages or dialects the user has to be familiar with. A radical way to achieve this is by replacing the various languages under discussion by a single language incorporating the characteristic features of them all. The main candidates for replacement are the command language, the programming language, the debugging language, and the editing language. To keep the scope of this study within reasonable bounds we have excluded editing languages from our considerations.

Although it may seem at first glance as if such an integration process would necessarily have to lead to a monstrous language incorporating all features ever invented by language and system designers, we hope to have shown that this is not the case. The reason is the conceptual inefficiency inherent in conventional systems. Most command language concepts are also present in programming languages although generally under a different guise. To a somewhat less extent the same is true for debugging concepts. Only because of this circumstance is language integration a meaningful proposition. Further integration of modes (especially of editing) may still be possible, but eventually it will be necessary to introduce sublanguage definition facilities into the basic language framework.

The designer of a unified command/programming/debugging language has less freedom than the designer of a conventional non-integrated language. Rather than being a disadvantage, the increased number of constraints helps him in settling design questions by offering various different - but equally valid - viewpoints from which to look at possible solutions. As long as they do not prevent 'the language equation' from being solved the language designer should welcome any additional criteria protecting him from the whims of his own taste.

The main features of monolingual environments as developed in this paper can be summarized as follows:

- ☐ No distinction is made between programs and procedures. Running a program is indistinguishable from calling a procedure; there is a single parameter mechanism throughout the system.
- ☐ No distinction is made between the types of permanent and local objects; there is no explicit input from or output to permanent objects.
- ☐ The type of every object is described by an abstract type definition. New type definitions may be added by the user. These may have either permanent or local status. Instances of permanent type definitions correspond to files in conventional systems.
- ☐ Procedures and type definitions are themselves objects that can be manipulated in the language.
- ☐ Permanent object directories and libraries do not have a special status, but are just one type of object definable by the user. This means that the structure of the environment is under control of the user. Since this is necessarily true at all levels, the data structures that are used as permanent object directories can also be used to structure the local environment. Of course, it is still possible to supply a predefined type *directory* or *library* with the system. A generalized PASCAL **with** statement serves to establish the current focus of interest, which may either be a directory type object or an object of any other type.

- ☐ An **interact** construct allows interactive programming at the point at which it occurs. The command level of the system corresponds to an implicit **interact** at the highest procedural level. Procedure and type declarations may (and probably will) be compiled, but if so, this must be completely transparent to the user.
- ☐ *Event associations* in conjunction with the **interact** construct mentioned in the previous point allow very selective interactive tracing without any source text modification. Event associations can also act as *production rules*, *exception handlers*, and *guards*.
- ☐ Automatic side-effect recovery in the form of **probe** and **try** constructs facilitates both the side-effect free inspection of (procedural) fields of objects in debugging mode as well as the programming of algorithms involving backtracking and error recovery. **undo** and side-effect free **assert** constructs make use of the same basic mechanism.
- ☐ Powerful string manipulation and pattern matching operations are provided. These can be looked upon as the forerunners of an integrated editing facility.

The greater conceptual simplicity of monolingual environments as compared with conventional systems hardly needs further amplification. Most existing APL and LISP environments are more highly integrated than conventional systems and do not have a separate command language. As a result, they do not make a distinction between the types of permanent and local data or between programs and procedures. The incorporation of other desirable features, such as system-wide abstract type definitions, is hampered by the fact that neither APL nor LISP allow user-defined types. The designers of SMALLTALK had greater freedom and based their system on an elaborated version of the SIMULA 67 class concept. In particular, SMALLTALK has system-wide class definitions and permanent class instances.

Similarly, most other features of monolingual environments have their counterpart in one system or another. We are not claiming originality in that respect. What is new, we believe, is the method we have sketched to bring these concepts together and to fuse them into a homogeneous whole.

ACKNOWLEDGMENTS

We are grateful to Martijn de Lange, Marleen Sint, Arthur Veen, and Paul Verhelst for their many helpful comments.

REFERENCES

The quotation at the beginning of this article was taken from: Weil, A., "De la métaphysique au mathématiques," in: *Collected Works*, Vol. II, Springer Verlag, 1980, pp. 408-412.

[ALB79] Alberga, C.N. , et. al., "A Program Development Tool," IBM, Thomas J. Watson Research Center, Report RC 7895 (#34000), 1979.

[ALL78] Allen, J., *Anatomy of LISP*, McGraw-Hill, 1978.

[BAY80] Bayer, M., et. al., "Software development in the CDL2 Laboratory," in: Hünke, H., (Ed.), *Software Engineering Environments*, North-Holland, 1980.

[BOU79] Bourne, S.R., "An introduction to the UNIX Shell," in: *UNIX Programmer's Manual*, Vol. 2A, Bell Telephone Laboratories, Inc., 7th Ed., 1979.

[BUR74] *APL/700 User Reference Manual*, Burrroughs Co., Pub. No. 5000813, 1974.

[BUR77] *B7000/B6000 Series Work Flow Language Reference Manual*, Burrroughs Co., Pub. No. 500155, 1977.

[BUX80] Buxton, J.N., "An informal bibliography on programming support environments," *SIGPLAN Notices*, **15**(1980), 12, pp. 17-30.

[BYT81] Special Issue on SMALLTALK , *BYTE*, **6**(1980), 8.

[DAK77] Davis, R., & King, J., "An overview of production systems," in: Elcock, E.W., & Michie, D., *Machine Intelligence 8*, Ellis Horwood Ltd., 1977, pp. 300-332.

[DOD80] *Ada Programming Language Military Standard*, Department of Defense, MIL-STD-1815, 10 December 1980.

[FEL79] Feldman, S.I., "Make - A program for maintaining computer programs," in: *UNIX Programmer's Manual*, Vol. 2A, Bell Telephone Laboratories, Inc., 7th Ed., 1979.

[FIV73] Falkoff, A.D., & Iverson, K.E., "The design of APL," *IBM Journal of Research and Development*, **17**(1973), 1, pp. 324-334.

[GKA76] Goldberg, A., & Kay, A., (Eds.), "SMALLTALK-72 Instruction Manual," XEROX Co., Palo Alto Research Center, Pub. No. SSL 76-6, 1976.

[GOO81] Goodwin, J.W., "Why programming environments need dynamic data types," *IEEE Transactions on Software Engineering*, **SE-7**(1981), 5, pp. 451-457.

[GPP71] Griswold, R.E., Poage, J.F., & Polonsky, I.P., *The SNOBOL4 Programming Language*, Prentice-Hall, 2nd Ed., 1971.

[GRI71] Grishman, R., "Criteria for a debugging language," in: Rustin, R., (Ed.), *Debugging Techniques in Large Scale Systems*, Courant Computer Science Symposium 1, Prentice-Hall, 1971, pp. 57-75.

[GRU79] Gruenberger, F.J., "The history of the JOHNNIAC," *Annals of the History of Computing*, **1**(1979), 1, pp. 49-64.

[HAN76] Hanson, D.R., "Variable associations in SNOBOL4," *Software Practice and Experience*, **6**(1976), pp. 245-254.

[HAN78] Hanson, D.R., "Event associations in SNOBOL4 for program debugging," *Software Practice and Experience*, **8**(1978), pp. 115-129.

[HUN80] Hünke, H., (Ed.), *Software Engineering Environments*, North-Holland, 1980.

[IBM78] *OS/VS2 TSO Command Language Reference*, IBM Co., Pub. No. GC28-0646-4, 5th Ed., June 1978.

[ING78] Ingalls, D.H.H., "The SMALLTALK-76 programming system - design and implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1978, pp. 9-16.

[JWI75] Jensen, K., & Wirth, N., *Pascal User Manual and Report*, Springer, 1975.

[KLI80] Klint, P., "An overview of the SUMMER programming language," *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, ACM, 1980, pp. 47-55.

[SAN78] Sandewall, E., "Programming in an interactive environment: the LISP experience," *Computing Surveys*, **10**(1978), 1, pp. 35-71.

[SHA64] Shaw, J.C., "JOSS: A designer's view of an experimental on-line computing system," *AFIPS Conference Proceedings*, **26**, *1964 Fall Joint Computer Conference*, pp. 455-464.

[TEI78] Teitelman, W., "INTERLISP Reference Manual," XEROX Co., Palo Alto Research Center, 1978.

[WIL80] Wilander, J., "An interactive programming system for PASCAL," *BIT*, **20**(1980), pp. 163-174.