SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 6, 245–254 (1976)

Variable Associations in SNOBOL4

DAVID R. HANSON

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

This paper describes a new facility in the SNOBOL4 programming language that provides the capability to associate the execution of a programmer-defined function with the act of assigning a variable a value or retrieving the value of a variable. The facility, called programmer-defined variable association, subsumes the existing built-in associations used for input, output and value tracing and provides, at the source-language level, the protection and datatype coercion mechanisms used in keywords. Several applications are described that illustrate the usefulness of this addition to the language. The facility is especially useful for program monitoring and debugging. The implementation of this facility and the effect of implementation techniques on programming language design are also discussed.

KEY WORDS SNOBOL4 Programming languages Implementation techniques Diagnostic facilities

INTRODUCTION

The SNOBOL4 programming language[1] contains certain operations that cause implicit actions whenever the value of a variable is referenced or a variable is assigned a value. For example, the statement

```
LINE = INPUT
```

usually results in the next line being read from the file associated with the variable INPUT and that value being assigned to LINE. The attempt to fetch the value of INPUT causes the initiation of an *implicit* internal process that reads the next line and assigns it to INPUT. Output in SNOBOL4 is performed in a similar fashion. Assignment of a value to the variable OUTPUT triggers an internal process that prints the value assigned on the output medium. This language feature makes input and output in SNOBOL4 very simple and is often cited by programmers as one of the virtues of using the language.

Implicit process activation is also evident in the value tracing facility. The statement

```
TRACE('X')
```

causes an internal process to monitor assignments to the variable X. Whenever a value is assigned to X, a message is issued indicating the new value of X. A notable feature of the value tracing facility is that tracing can be turned off and on at the programmer's discretion in the source language.

SNOBOL4 is a language without type declarations for variables. However, keywords are an instance where the valid types of a variable must be restricted. This restriction is handled whenever a value is assigned to keyword. For example, the assignment

```
&STLIMIT = '10000'
```

causes an internal process to intervene in order to insure the value assigned to &STLIMIT is of datatype INTEGER.

Received 6 May 1975

In each of the examples above, the act of referencing the value of a variable or assigning a value to a variable results in the intervention of a specific SNOBOL4 internal process. This paper describes an addition to the SNOBOL4 language that allows the programmer to associate the act of referencing a variable with a programmer-defined function in a fashion similar to that which is used for the existing input, output and trace associations. The concept of the programmer-defined association provides a basis for unification of this type of language feature and elevates a useful implementation technique to a source-language construct. The facility subsumes the existing built-in associations for input, output and value tracing. In addition, it is completely dynamic in that the nature of the association can be changed at any time and a single variable may have any number of associations.

Initial use of this new language feature indicates that it is well-suited for applications such as datatype coercion, data structure access and manipulation, generators,[2] program monitoring, and especially program debugging.[3]

VARIABLE ASSOCIATION

Associations

There are two kinds of associations. Associations such as the built-in input association are referred to as *fetch* associations, since they are triggered whenever the value of a variable is fetched. The type of association exhibited in value tracing is called a *store* association. A programmer-defined association may be either a fetch or store association. A variable may have any number of associations of either type. An association is made by the execution of a statement of the form

ASSOCIATE(*name*, *processdescription*)

ASSOCIATE is a built-in function. The *name* is the name of the variable to be associated. The *processdescription* is an instance of a programmer-defined data object with four fields. The contents of the fields describe the process that is associated with the variable given by *name*. The programmer-defined data object that is usually used is defined by the statement

DATA('PROCESS(TYPE, FUNCT, ACTIVE, LEVEL)')

The TYPE field must be either the string 'FETCH' or 'STORE', indicating the type of association. The FUNCT field contains the name of the built-in or programmer-defined function that is to be called whenever the value of the associated variable is stored or fetched (depending on the contents of the TYPE field). If the ACTIVE field contains a non-zero integer when the variable is referenced for either a fetch or store, the association is considered active and the function named in the FUNCT field is called. If the value is zero, the association is ignored. In the case where there are multiple associations for a single variable, the integer contained in the LEVEL field is used to indicate the position of the association with respect to the existing associations with the variable.

A programmer-defined association function is called with three arguments. For a fetch association, the arguments are;

(i) the name of the associated variable,
(ii) the current value of that variable,
(iii) the process description.

For example, if the variable Y is fetch-associated by the execution of the statement

ASSOCIATE('Y', PROCESS('FETCH', 'F', 1, 1)) then the semantics of the statement

```
Z = Y
```

are equivalent to the standard SNOBOL4 statement

```
Z = F('Y', Y, PROCESS('FETCH', 'F', 1, 1))
```

The value returned by the function F is assigned to Z. Note that F need not return the current value of Y and may succeed or fail. In the latter case the statement fails and no assignment to Z is made.

For a store association, the first and third arguments are the same as for a fetch association. The second argument is the value to be assigned. The value returned by the association function is ultimately assigned to the variable. Thus the association function can intervene during assignment and change the value to be assigned or even prevent assignment by failing. For example, if X is store-associated with a process that has an association function G, then the semantics of the statement

```
X = 'DOG'
```

are equivalent to

X = G('X', 'DOG', *processdescription*)

If G fails, no assignment is made.

When an association function is called, the association that caused the function to be invoked is deactivated until the function returns. Other associations with the same variable remain active, however, so that a structurally simple statement can trigger many function calls.

As an example, the existing value-tracing facility provided in SNOBOL4 can be implemented easily using programmer-defined associations.

```
           TPROCESS = PROCESS('STORE', 'TRACER', 1000, 1)

           DEFINE('TRACE(NAME)')                          :(TRACE.END)
TRACE      ASSOCIATE(NAME, TPROCESS)                      :(RETURN)
TRACE.END

           DEFINE('TRACER(NAME, TRACER, TPROCESS)')       :(TRACER.END)
TRACER     OUTPUT = 'Statement ' &LASTNO ': ' NAME ' = '
                    TRACER ', Time = ' TIME()
+          ACTIVE(TPROCESS) = ACTIVE(TPROCESS) - 1        :(RETURN)
TRACER.END
```

The association function TRACER simply returns the value that is about to be assigned after issuing a standard trace message. The ACTIVE field of TPROCESS is used in an analogous fashion to the keyword &TRACE; it is decremented by one at every intervention until it reaches zero, at which point the association becomes inactive. The programmer can re-activate the association merely by setting the value of the ACTIVE field to a non-zero integer. Many variables can have the same associated process description so that the modification of the ACTIVE field affects all associations with TPROCESS. By using a fetch association, the programmer can also implement a facility to issue a message whenever the value of a variable is referenced.

Assuming the existence of primitive, built-in functions READ and WRITE, the existing input and output associations can also be implemented using fetch and store associations in a similar manner.

Removing associations

The built-in function

DISASSOCIATE(*name*, *processdescription*)

undoes the effect of ASSOCIATE—all the associations with the variable given in *name* and described by *processdescription* are removed. Other associations are not affected.

In addition to the ACTIVE field in the process description, the keyword &ASSOCIATE controls the activation of all associations. If &ASSOCIATE is zero, all associations are ignored regardless of the values of the individual ACTIVE fields. A non-zero value causes the ACTIVE fields to be checked to determine if an association is active or inactive. Initially &ASSOCIATE is 1.

APPLICATIONS

Generators

Programmer-defined associations can be used to cause a reference to a variable to act as a generator function. For example, most compilers have a function that generates unique labels. The following program segment associates a process with the variable NXTLAB so that every attempt to fetch the value of NXTLAB causes the generation of a unique label for the value of NXTLAB.

```
            ASSOCIATE('NXTLAB', PROCESS('FETCH', 'GENLAB', 1, 1))
            DEFINE('GENLAB(NAME, CURRENT)')        :(GENLAB.END)
GENLAB      $NAME = CURRENT + 1
            GENLAB = 'L' $NAME                     :(RETURN)
GENLAB.END
```

This function generates the sequence of labels L1, L2, L3, . . ., Ln. Note that GENLAB may be associated with any number of variables simultaneously and cause references to those variables to generate independent sequences of labels. There is no explicit reference to NXTLAB in GENLAB binding that function to that particular variable. NXTLAB is referenced indirectly by the use of standard SNOBOL4 indirect referencing operator, $. A similar association can be used to cause a reference to the value of a variable to return a pseudo-random number, the next character in a string, etc. Similarly the built-in input association causes references to INPUT to act like a generator that generates the next line of input text.

Datatype coercion

A variable in SNOBOL4 can have a value of any datatype at any time during program execution. This feature is usually valuable, but in some circumstances may lead to errors. Programmer-defined variable associations can be used to provide a protection or datatype coercion mechanism similar to that used for keywords. For example, suppose it is desired that a variable always have objects of a particular datatype as value. If a value is assigned to the variable, it must be of the correct type or convertible to it. Using programmer-defined store associations and a TABLE, we can write a function DECLARE to produce the desired result.

```
             TYPETAB = TABLE()
             DTPROCES = PROCESS('STORE', 'DTCHEK', 1, 1)

             DEFINE('DECLARE(NAME, TYPE)')               :(DECLARE.END)
DECLARE      DISASSOCIATE(NAME, DTPROCES)
             TYPETAB<NAME> = TYPE
             ASSOCIATE(NAME, DTPROCES)                   :(RETURN)
DECLARE.END

             DEFINE('DTCHEK(NAME, DTCHEK)LN')            :(DTCHEK.END)
DTCHEK       LN = &LASTNO
             LEQ(DATATYPE(DTCHEK), TYPETAB <NAME>)
+                                                        :S(RETURN)
             DTCHEK = CONVERT(DTCHEK, TYPETAB <NAME>)
+                                                        :S(RETURN)
             OUTPUT = 'Statement ' LN ': Attempt to assign '
+                     'wrong datatype to ' NAME
+                                                        :(FRETURN)
DTCHEK.END
```

The function DECLARE can be called at any point during program execution to dynamically restrict a variable to values of a specified type, e.g.

```
                     DECLARE('COUNT', 'INTEGER')
```

Datatype checking can be globally deactivated by setting the value of ACTIVE(DTPROCES) to 0.

Data structure manipulation

Automatic data structure access and manipulation provide an intriguing use of associations. For example, the operations of pushing and popping values on and off a stack are used in many applications. The following program segment provides associations that push a value whenever an assignment is performed and pop the stack whenever the value of the associated variable is referenced.

```
             DATA('PLATE(VALUE, LAST)')
             ASSOCIATE('P', PROCESS('STORE', 'PUSH', 1, 1))
             ASSOCIATE('P', PROCESS('FETCH', 'POP', 1, 1))
             DEFINE('PUSH(NAME, PUSH, SP)')              :(PUSH.END)
PUSH         PSTACK = PLATE(PUSH, PSTACK)                :(RETURN)
PUSH.END

             DEFINE('POP(NAME, X, SP)')                  :(POP.END)
POP          POP = DIFFER(PSTACK) VALUE(PSTACK)          :F(FRETURN)
             PSTACK = LAST(PSTACK)                       :(RETURN)
POP.END
```

The variable P is associated so that assignments and value references to P trigger the PUSH and POP functions. The actual stack is a linked list of programmer-defined data objects of datatype PLATE beginning with PSTACK. Note that the actual value of P is ignored. P is used only to cause the appropriate function, PUSH or POP, to be called.

Program monitoring and debugging

Programmer-defined associations are especially well-suited for use as an aid to program debugging and monitoring. As illustrated in the value tracing and datatype coercion examples given above, the programmer can write functions to monitor the access of values, to make certain variables read-only or write-only, and enforce desired datatype and value restrictions. Associations can be made not only with natural variables but also with created variables such as array and table elements and fields of programmer-defined data objects.

As a further example, suppose the programmer wishes to monitor the addition of items onto the end of a queue implemented as a singly-linked list of programmer-defined data objects. The elements of the queue can be defined by

```
DATA('NODE(VALUE, LINK)')
```

The queue is initialized by the statements

```
HEAD = NULL
PTR = .HEAD
```

An entry is added to the queue by the statements

```
$PTR = NODE(item)
PTR = .LINK($PTR)
```

Monitoring the addition of NODEs onto the end of the list beginning with HEAD can be done by using a store association with the LINK field of the NODE that is filled when the next entry is added. Thus the association 'moves' with the end of the list.

```
           LPROCESS = PROCESS('STORE','LISTMTR', 1, 1)
           DEFINE('LISTMTR(NAME, LISTMTR)')          :(LISTMTR.END)
LISTMTR    OUTPUT = 'Stmt '&LASTNO ': ' VALUE(LISTMTR)
+                   ' added to list'
           DISASSOCIATE(NAME, LPROCESS)
           ASSOCIATE(.LINK(LISTMTR), LPROCESS)       :(RETURN)
LISTMTR.END
```

The statement

```
ASSOCIATE(.HEAD, LPROCESS)
```

must be executed initially to begin monitoring additions to the list.

Multiple associations

In the case of multiple associations, store associations are executed in the order of the process level numbers (the LEVEL field of the process description), whereas fetch associations are executed in the reverse order. It is usually desired that programmer-defined associations used for monitoring purposes have their functions executed first when there are multiple associations. To insure this, the programmer can assign a small level number to store associations and a large level number to fetch associations. The assignment of level numbers in other cases is only important if two associations can interact, e.g. in preprocessing strings assigned to OUTPUT. The default level numbers for built-in associations are 100 for value tracing, 200 for output association and 300 for input association. For example, to produce a page header every 66 lines of output produced by assignment to OUTPUT, the store association must be given a level number less than 100. The following program segment accomplishes the desired result.

```
           LINENO = 66
           PAGENO = 0
           ASSOCIATE('OUTPUT',PROCESS('STORE','HEADER',1,50))
           DEFINE('HEADER(NAME,HEADER)')        :(HEADER.END)
HEADER     LINENO = LT(LINENO,66) LINENO + 1    :(RETURN)
           PAGENO = PAGENO + 1
           OUTPUT = 'Page '  PAGENO
           OUTPUT =
           LINENO = 3                           :(RETURN)
HEADER.END
```

During the execution of the function HEADER, the association is deactivated so that assignments to OUTPUT trigger only the built-in output association.

Discussion

Programmer-defined associations allow the programmer to associate an arbitrarily complex process with the simple act of referencing a variable. This is the type of facility that is needed in program monitoring and debugging. Variable association makes possible the addition of monitoring functions without having to modify the text of the program itself. One application in this area has been the development of a programmable, general-purpose debugging package for SITBOL that is coded in the source language.[3]

The SNOBOL4 programmer can use programmer-defined associations as an aid to the top-down development of SNOBOL4 programs. For example, a first step can be to associate the variable OUTPUT with the built-in process for writing strings to a listing file. A subsequent refinement might be made to provide page headings or special formatting, using another store association with the variable OUTPUT. The latter step can be done without the explicit knowledge of every assignment to OUTPUT or extensive reprogramming to implement the desired feature. Moreover, associations are guaranteed to affect all references to a variable regardless of the context in which they are used. This is important in a language that has constructs that can cause 'hidden references' to a variable such as indirect referencing or execution-time compilation.

Although the notion of variable association has been applied only to SNOBOL4, the concept is applicable to other languages in a similar fashion. For an interpreter-based implementation of a programming language such as SNOBOL4, APL or LISP, the dynamic character of variable associations is not difficult to achieve. In languages such as FORTRAN, PL/I and ALGOL60, whose implementations are compiler-based, a variable association facility most likely must be a static construct bound at compile-time. However, this constraint does not seriously restrict the usefulness of variable associations as described above. Experience thus far indicates that the use of variable associations falls into two major categories—associations of a static nature used as a general programming technique such as providing page headings and dynamic variable associations used for program monitoring and debugging.[3] Even a dynamic association is most often used in a manner where the activation of the association is the only dynamic component. There is little reason why variable associations of this kind cannot be included in a programming language that has a compiler-based implementation.

IMPLEMENTATION

The implementation of programmer-defined variable associations in a manner that does not have an adverse impact on system performance is a non-trivial problem. A feature of this type is sometimes discarded during language design because the apparent implementation method is too inefficient or unwieldy. To correctly implement variable associations in SNOBOL4, a check for an associated variable must be made *before* every fetch and store in a reasonably efficient fashion. The existing facilities of input and output association and value tracing impose a similar requirement. In the macro implementation of SNOBOL4,[4] the responsibility of checking for input and output association and value tracing rests with the operation of assignment and value fetching. At every fetch and store operation, a table is searched to determine whether the given variable is traced or associated for input or output. The time required for assignment or value fetching for any variable is proportional to the number of associated variables. In SITBOL,[5-7] a full SNOBOL4 interpreter implementation for the DEC-10, the burden of an association is placed on the variable rather than the operation. In this case, the time required to store or fetch a value is independent of the number of associated variables and this method is significantly more efficient than the one used in the macro implementation.

The SITBOL mechanism consists of replacing the value of an associated variable by a 'trapped variable. [5,8] Values in SITBOL are represented by 'descriptors' consisting of two DEC-10 36-bit words containing a datatype indicator, several system flags and the actual value or a pointer to a block in the allocated data region[5] housing the value. A trapped variable descriptor consists of a pointer to a trapping block and an address of a procedure that is to handle the trap. The trapping block contains the original value plus any information necessary to process the trap. Trapped variables are used internally in SITBOL not only for value tracing and input and output associations, but also for keywords, some aspects of pattern matching and table references. References to a variable in order to store or fetch a value are performed by two system procedures that check for trapped variables and call the appropriate trapping procedure if necessary.

The implementation of programmer-defined associations is a natural extension of the use of trapped variables. The ASSOCIATE function replaces the value of the associated variable with a trapped variable that points to a trapping block containing the original value and a pointer to the process description. It also contains the address of a trapping procedure to handle the invocation of programmer-defined association functions. The original value housed in the trapping block can also be a trapped variable, thereby implementing a chain of traps of arbitrary length. The DISASSOCIATE function simply disconnects the trapping block by overwriting the trapped variable with the original value. Further details concerning the implementation scheme are given in References 5 and 9.

CONCLUSIONS

SNOBOL4 is a prime example in which programming language implementation has had a significant and beneficial effect on language design. Features such as execution-time compilation, unevaluated expressions, keywords and, to some extent, the cursor assignment operator were originally suggested by the implementation or the desire to elevate a useful implementation construct to the source language.[4] The effect of the underlying implementation on the evolution of the keyword facility is particularly evident. In the macro implementation, references to keywords simply return, or assign a value to, a system status variable defined by the implementation. In later implementations,[5-7, 10-12] some keyword references are used to trigger internal processes that perform more complex actions than described above. For example, in SITBOL, one of the more recent SNOBOL4 implementations, the assignment of a positive integer to the keyword &HISTOGRAM causes the initiation of an internal process that constructs and supervises the maintenance of an array containing a time histogram of the program behaviour on a per-statement basis. This extension of the keyword concept is partially due to the development of more sophisticated implementation techniques such as the trapped variable. The trapped variable scheme provides a unified and efficient method for the implementation of the seemingly disjoint constructs of keywords, input and output associations and value tracing. The programmer-defined variable association facility itself is the result of the elevation of the trapped variable mechanism to the source-language level for use as a general programming tool.

In the later stages of this work, it became apparent that a fundamental concept in variable associations was to allow the programmer to intervene, *via* a programmer-defined function, on the occurrence of a specified event; namely fetching or storing a value of a specific variable. This suggests that a variable association is one, well-defined manifestation of a more general *event association*. Research is currently in progress to investigate a facility of programmer-defined associations with members of a set of predefined, 'built-in' events and programmer-defined events. Programmer-defined associations with events such as statement execution and interruption,[3] specific errors, function call and return, operator invocation, time and environment dependent functions, and internal events such as storage allocation are being studied. Initial results[3] indicate that the general event association concept is both useful and powerful but that the conceptualization and implementation is significantly more difficult than for programmer-defined variable associations.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the helpful comments of many of my colleagues while working on this subject. Special thanks are due to Frederick C. Druseikis who helped me 'discover the true nature of the beast' and Ralph E. Griswold for his suggestion that associations be described by the contents of a programmer-defined data object rather than by additional arguments to the ASSOCIATE function.

REFERENCES

1. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd. edn, Prentice-Hall, Englewood Cliffs, N.J., 1971.
2. R. E. Griswold, *String and List Processing in SNOBOL4: Techniques and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1975, chps 2 and 7.
3. D. R. Hanson, *Additions to the SITBOL Implementation of SNOBOL4 to Facilitate Interactive Debugging*, SNOBOL4 Project Document S4D51, The University of Arizona, Tucson, 1975.
4. R. E. Griswold, *The Macro Implementation of SNOBOL4, A Case Study of Machine-Independent Software Development*, W. H. Freeman, San Francisco, 1972.
5. J. F. Gimpel, *A Design for SNOBOL4 for the PDP-10*, SNOBOL4 Project Document S4D29b, Bell Laboratories, Holmdel, N.J., 1973.
6. J. F. Gimpel, *SITBOL Version 3.0*, SNOBOL4 Project Document S4D30b, Bell Laboratories, Holmdel, N.J., 1973.
7. J. F. Gimpel, 'Some highlights of the SITBOL language extensions to SNOBOL4', *SIGPLAN Notices*, **9**, 11–20 (1974).
8. J. F. Gimpel and D. R. Hanson, *The Design of ELFBOL—A Full SNOBOL4 for the PDP-11*, SNOBOL4 Project Document S4D34, Bell Laboratories, Holmdel, N.J., 1973.
9. D. R. Hanson, *Programmer-Defined Variable Associations in SNOBOL4*, SNOBOL4 Project Document S4D50, The University of Arizona, Tucson, 1975.

10. R. B. K. Dewar, *SPITBOL Version 2.0*, SNOBOL4 Project Document S4D23, Illinois Institute of Technology, Chicago, 1971.
11. P. J. Santos, Jr., *FASBOL, A SNOBOL4 Compiler*, Electronics Research Laboratory Memorandum No. ERL–M314, The University of California, Berkeley, 1971.
12. P. J. Santos, Jr., *FASBOL II, A SNOBOL4 Compiler for the PDP-10*, Electronics Research Laboratory Memorandum No. ERL–M348, The University of California, Berkeley, 1972.