Mitchell H. Clifton

Self-Assessment Procedure XXIII: Programming Languages

This is the 23rd Self-Assessment Procedure.[1] The purpose of this procedure is to let its readers test their knowledge of some of the more important features of significant programming languages. The features of imperative languages like Fortran, Algol 60, PL/I, Pascal, Modula-2, C, and Ada considered are data types, data structures, control structures, program units, scope of names, and parameter passing. Some of the corresponding features of functional languages like Lisp, logic programming languages like Prolog, and ohject-oriented programming languages like C++ and Smalltalk, are reviewed. Finally brief consideration is given to syntax and semantics, chiefly by reference to the Backus-Naur form, BNF. The features of programming languages affect all aspects of programming, including the ease of writing and maintaining programs, the reliability of software, and the efficiency and portability of programs. Knowledge of programming language concepts and constructs is of importance to software developers, programmers, and designers of programming languages and compilers. The topic of this procedure should be of interest to any computer science student or practitioner who wishes to test his or her knowledge and learn more about one of the most important and broad areas in the computer field. The language used most often in the questions is Pascal, with Ada, Lisp, and Prolog also featured prominently. A glossary is included which defines some of the terms used in the procedure. The questions are either multiple-choice or short answer. Some of the multiple-choice questions allow for more than one correct answer.

Self-Assessment Procedures and How to Use Them

Self-assessment is based on the idea that a procedure can be devised that will help a person appraise and develop his or her knowledge about a particular topic. It is intended to be an educational experience for a participant. The questions are only the beginning of the procedure. They are developed to help the participant think about the concepts and decide whether to pursue the matter further.

The primary motivation of self-assessment is not for an individual to satisfy others about his or her knowledge; rather it is for a participant to appraise and develop his or her own knowledge. This means there are several ways to use a self-assessment procedure. Some people will start with the questions. Others will read the answers and refer to the references first. These approaches and others devised by the participants are all acceptable if at the end of the procedure the participant can say, "Yes, this has been a worthwhile experience" or "I have learned something."

We suggest the following way of using the procedure, but as noted earlier, there are others. This not a timed exercise; therefore, plan to work with the procedure when you have an hour to spare, or you will be shortchanging yourself on this educational experience. Go through the questions, and mark the responses you think are most appropriate. Compare your responses with those suggested by the Committee. In those cases where you differ from the Committee, look up the references if the subject seems pertinent to you. In those cases in which you agree with the Committee, but feel uncomfortable with the subject matter, and the subject is significant to you, look up the references.[2]

Some ACM chapters may want to devote a session to discussing this self-assessment procedure or the concepts involved.

The Committee hopes some participants will send comments.

Part I. Questions

Data Types and Structures

Programming languages have standard data types (integer, real, and character) and data structures (arrays, records, and lists). For each data type and structure a way to map data to storage must be provided. This section has questions about data types and structures, their implementation, and type checking

1. Below is the declaration of a variable in Pascal.

   var I : Integer;

   The size of storage allocated for the variable I is determined:

   a. Dynamically, by the data types of the values assigned to I at run-time.
   b. Dynamically, by the sizes of the values assigned to I at run-time.
   c. Statically, by the allowed range of integers in the Pascal language definition.
   d. Statically, by the integer representation of the underlying hardware.

2. Most programming languages allocate storage for arrays sequentially and require all array elements to be of the same type, to allow efficient calculation of storage addresses of elements at run-time. Suppose the Pascal array below is allocated sequentially in storage, in row-major order, with 4 bytes per element, and starting address b.

   var A : **array**[1 .. 5, 1 .. 10] **of** Real;

   The expression to calculate the storage address, in bytes, of element A[I, J] is:

   a. b + 4 * (I − 1) + 20 * (J − 1).

[1]See box at end of article for a list of the previous self-assessment procedures (SAPs). The first 13 SAPs are available from the ACM Order Department, 1-800-342-6626 (in U.S. and Canada); 1-212-626-0500 (metro NY and outside U.S.); +32-2-774-9602 (in Europe); acmhelp@acm_org; acm europe@acm.org (in Europe). Members; $22; nonmembers; $43. Order number 203840.

[2]Sap is not sanctioned as a test or endorsed in any way by ACM. Any person using any of the questions in this procedure for the testing or certification of anyone other than him- or herself is violating the spirit of this self-assessment procedure and the copyright on this material.

b. b + 40 * (I − 1) + 20 * (J − 1).
c. b + 40 * I + 4 * J − 44.
d. b + 20 * I + 4 * J − 24.

3. Pascal has a data structure called a variant record, an example of which is shown below. A common implementation of this data structure is to overlap the variant parts of the record in storage.

```
var Item : record
        A : Integer;
        case B : Boolean of
          False: (C, D, E, F : Char);
          True: (G : Real)
        end;
```

Although Pascal has many characteristics of a secure language, variant records introduce a potential security loophole. This example illustrates a security loophole in Pascal because:
   a. The variable A is not in a variant part of the record.
   b. It is unclear whether the variables C-F take the same amount of storage as G.
   c. On many systems, it allows a real value assigned to the variable G to be interpreted as characters.
   d. On many systems, an assignment of a value to the variable C destroys any value assigned to G.

4. Use of enumerated data types may lead to more secure and readable programs and save storage. Below are declarations of an enumerated data type and variable in Pascal.

```
type GradeType = (A, B, C, D, E, H, I, P, W);
  var Grade : GradeType;
```

The minimum storage space needed for the variable Grade is:
   a. 3 bits.
   b. 4 bits.
   c. 9 bits.
   d. 9 bytes.

5. Programming languages provide *coercions* between different data types. Below are declarations of two variables in Pascal.

```
var I : Integer;
    X : Real;
```

The Pascal assignment statement that shows the use of a coercion is:
   a. X := X + I;
   b. X := 2.5 * X;
   c. I := Round (X);
   d. I := I + Ord ('0');

6. Languages such as Pascal and Ada require the variable on the left hand side of an assignment statement, and the expression on the right hand side, to be of the same type. Two ways of determining if they are the same type are *structural equivalence* and *name equivalence*. Which of the following are reasons that languages like Ada and ISO Standard Pascal have adopted a form of name equivalence for type checking?
   a. Name equivalence is more restrictive, and hence more secure.
   b. A precise definition of structural equivalence is not possible.
   c. Use of name equivalence makes the language more convenient for the programmer.
   d. Name equivalence is simpler to implement.

7. Type checking may be *static* (performed at translation time) or *dynamic* (performed at run-time). Which of the following are true about type checking?
   a. Use of dynamic type checking, instead of static type checking, allows faster program execution.
   b. Program correctness is enhanced by a language that requires type checking.
   c. Dynamic type checking is used more by interpreted languages than by compiled languages.
   d. Static type checking cannot be fully implemented if separate compilation of program modules is allowed.

8. The run-time storage space for a Pascal program usually includes a static allocation area, a stack, and a heap. Which of the following features of Pascal require the use of the heap?
   a. Pointers and the New and Dispose procedures for creating new data elements.

Glossary

Coercion: The automatic, or implicit, conversion of one data type to another.
Control structure: A language construct that prescribes the order in which expressions and statements are executed.
Data structure: An aggregate of data objects.
Data type: A class of data objects together with a set of operations for manipulating them.
Functional language: A language in which function application is the basic unit of program construction.
Imperative language: A language characterized by many computations and assignments on small items of data, with data and control structures that are similar to a machine architecture.
Logic programming language: A language in which programs are assertions in a logic, and computation corresponds to proving or satisfying the assertions.
Overlap: To have one data structure use the same memory locations as another data structure.
Parameter passing: The transfer of arguments to a procedure or function.
Program unit: The building block of a program.
Scope of names: The range over which a name is known.
Semantics: The rules that specify the meaning of syntactically valid constructs in a language.
Side effect: Any change in state beyond the simple returning of a value.
Space efficiency: The effectiveness with which memory is utilized.
Syntax: The rules that specify the form of valid constructs in a language.

b. Recursive procedures.
c. Variant records.
d. Arrays.

Control Structures

Programming languages have mechanisms to control the flow of execution of statements and expressions. Some languages provide structures for concurrent or parallel execution of program units. This section has questions about statement-level, expression-level, and concurrent programming control structures.

9. Which of the following control statements are often implemented using a jump table?
   a. **Case** statements.
   b. **For** statements.
   c. Nested **if** statements.
   d. **While** statements.

10. Below is an example of a counter-controlled loop in Pascal.

```
for I := 1 to N do
    Sum := Sum + A[I];
```

There are many design issues for counter-controlled loops. Which of the following questions have been addressed the same way in the design of counter-controlled loops in Fortran, Algol 60, Pascal, C, and Ada?
   a. What is the scope of the loop control variable?
   b. Is it valid to branch into or out of loops?
   c. Is it valid to nest loops?
   d. Can the loop control variable be changed inside the loop, and if so, does the change affect loop control?

11. Consider this Pascal program:

```
program Main;
    var  A, B, X : Integer;
    function F(N : Integer) : Integer;
    begin
        B := 10;
        F := N + 2
    end;
begin
    A := 4;
    B := 5;
    X := F(A) + A * B;
    Writeln (X)
end.
```

Pascal does not completely define the order of evaluation of expressions. In the assignment statement for X, the language does not specify whether the function F is performed before or after the multiplication. Usually, this order is not important and has no effect on the results. However, if the function causes a *side effect,* as the function F does, the order of evaluation is important.

Due to this side effect, and the incomplete definition in Pascal of the order of evaluation, what are all possible values of X that might be computed and output by different Pascal systems?

12. Below is an example of an **if** statement in C to increment the value of the variable count by one if both *exp1* and *exp2* are true (++ is the C increment operator and && is the boolean **and** operator).

```
If ( exp1 && exp2 )
    count++;
```

C specifies the *short-circuit evaluation* of expressions involving boolean **and** and **or** operators. Which of the following are possible advantages of the short-circuit evaluation of the expression in this example?
   a. Less machine code will be compiled.
   b. The expression may execute faster, since *exp2* is evaluated only if *exp1* is true.
   c. The evaluation of *exp1* may be used to guard against a possible run-time error in the evaluation of *exp2*.
   d. Side effects caused by the evaluation of *exp2* will not be lost.

13. Most programming languages designed since Lisp and Algol 60 have provided recursive functions and procedures. Below is an example of a recursive function in Pascal.

```
function F (N : Integer) : Integer;
begin
    if N > 0 then
        F := 2 * F(N − 1)
    else
        F := 1
end;
```

What is the expression computed by the evaluation of F(N), for N >= 0?

14. Programming languages such as Ada and PL/I provide *exception handlers,* to handle special run-time events, called *exceptions.* Which of the following are usually true about exceptions and exception handlers?
   a. Exceptions may be generated by execution of statements provided just for that purpose.
   b. Exceptions may be generated by run-time errors, such as arithmetic overflow or an array subscript out of bounds.
   c. Exception handlers are structured similar to procedures, including a list of parameters, a set of declarations, and a sequence of statements.
   d. If an exception is generated in a block or procedure, and no exception handler is defined in the block or procedure for the exception, then the program is terminated.

15. Some programming languages use a **cobegin/coend** statement to initiate the concurrent execution of statements. The example below shows the concurrent execution of two assignment statements.

```
X := 2;
cobegin
    X := X + 3;
    X := X + 1
coend
```

When concurrently executing statements access shared variables, indeterminate results may occur. What are all possible final values for the variable X when the **cobegin/coend** statement finishes?

16. Below is part of an Ada program to initiate the concurrent execution of two processes, called *tasks* in Ada. When procedure P1 is called, task T1 is automatically started, resulting in the concurrent execution of P1 and T1. Assume that no input/output operations are started by the execution of P1 and T1.

```
procedure P1 is
    task T1;
    task body T1 is
    begin
        . . task T1 statements . .
    end T1;
begin
    . . procedure P1 statements . .
end P1;
```

The statement that best describes the execution of this program on a system having a single processor is:
   a. A run-time error will result, since this program must be run on a system with at least two processors.
   b. The program will execute faster than an equivalent program that does not use tasks.
   c. The program will not execute faster than an equivalent program that does not use tasks, so there is no reason to use tasks if the program will only be run on a single processor system
   d. Even though the program will not execute faster than an equivalent program that does not use tasks, the use of tasks in the program may still represent the most natural solution to a problem.

17. Ada tasks may interact through a type of procedure call called a *rendezvous*. The best description of the type of interaction provided between Ada tasks by the rendezvous is:
   a. Communication and mutual exclusion.
   b. Communication and synchronization.
   c. Mutual exclusion and deadlock prevention.
   d. Deadlock prevention and synchronization.

Names and Data Control

Programming languages divide programs into units (such as procedures, packages, and modules), and control the scope of names and data with these units. The questions in this section concern program units, the scope of names, and parameter passing.

18. An important point in name control is whether *static scope* (also called lexical scope) or *dynamic scope* is used to locate names that are not local to a block or procedure. Which of the following are true about static and dynamic scope?
   a. Static scope binds names to declarations based on the static structure of the program; dynamic scope binds names to declarations at run-time.
   b. Dynamic scope is used by languages such as C and Pascal.
   c. With dynamic scope complete type checking cannot be performed until run-time.
   d. Use of dynamic scope instead of static scope leads to programs that are easier to understand.

19. Below is a Pascal-like program with procedures P1, P2, and P3. The main program calls procedure P1, P1 calls P3, and P3 calls P2. In procedure P2 the value of the variable I is written.

```
program Main;
    var I : Integer;
    procedure P1;
        var I : Integer;
        procedure P2;
        begin
            Writeln (I)
        end; {P2}
        procedure P3;
            var I : Integer;
        begin
            I := 5;
            P2
        end; {P3}
    begin
        I := 7;
        P3
    end; {P1}
begin
    I := 3;
    P1
end. {Main}
```

What value will be written by the program if the following scope rules are used to locate variable I?
   1. Dynamic scope.
   2. Static, or lexical, scope.

20. The Pascal-like procedure below is used to show the effects of different parameter passing methods.

```
procedure Test (A, B, C : Integer);
begin
    A := A + 1;
    Writeln (B, C)
end;
```

The procedure is called after an assignment to the integer variable I with the statements below.

```
I := 5;
Test(I, I, 2*I);
```

What values will be written for B and C, using the following parameter passing methods?
   1. Pass by value-result.
   2. Pass by reference.
   3. Pass by name.

21. The activation record for a procedure in Pascal does *not* contain:
   a. Values or addresses of parameters.
   b. Values of local variables.
   c. Procedure code.
   d. The address of the return point.

22. Language constructs such as Modula-2 modules, Ada

packages, and object-oriented programming classes permit the *encapsulation*, or combination, of data and its associated operations, allowing the creation of abstract data types. Which of the following are benefits of the encapsulation of data and operations?

a. Programs store data more efficiently.
b. Programs perform operations more efficiently.
c. Programs are easier to read and understand.
d. Programs are easier to maintain.

23. Modules, packages, and classes allow data and operations to be declared to be either *public* or *private*. Suppose a module is used to construct an abstract data type for a queue. The module includes declarations for the queue indexes and procedures to insert and remove elements. The most appropriate declarations for these queue indexes and procedures would be to declare that:

a. Both indexes and procedures are private.
b. Indexes are private and procedures are public.
c. Indexes are public and procedures are private.
d. Both indexes and procedures are public.

Functional, Logic, and Object-Oriented Programming

The prior sections of this self-assessment have been concerned with imperative, or conventional, programming languages. In imperative languages, data and control structures are closely related to a computer architecture. This part is concerned with other styles of programming and programming languages, including functional programming and the Lisp language, logic programming and the Prolog language, and object-oriented programming.

24. Lisp has many of the characteristics of a functional or applicative language. However, one Lisp function that is *not* representative of a purely functional language is:

a. CAR, the function to extract the first element of a list.
b. COND, the conditional test function.
c. PLUS, the addition function.
d. SETQ, the assignment function.

25. In many versions of Lisp, including the Scheme dialect, functions are *first-class values* and may be *higher order*. Which of the following can be done with first-class and higher order functions in Lisp?

a. Functions can be passed as arguments to functions.
b. The value returned by a function can be a function.
c. Function arguments can be statically type checked.
d. Functions can be stored as elements of lists.

26. Lisp introduced a number of important concepts, including *garbage collection* which reclaims previously allocated storage that is no longer in use by programs. Which of the following are true concerning garbage collection in Lisp systems.

a. It ensures that programs will not run out of storage.
b. It may be performed when free storage is exhausted, or when only a certain fraction of free storage remains to be allocated.
c. It requires negligible time.
d. It may not reclaim all unused storage, if a programmer has not deallocated storage properly.

27. Lisp has been widely used for applications in artificial intelligence, but has been far less popular for other applications. Which of the following concerning Lisp are true, and have caused the language to be less popular?

a. The major data structure in Lisp, the list, is inflexible.
b. Lisp systems have often lagged behind other programming language systems in providing adequate programming environments, including editors and debuggers.
c. Lisp interpreters have often been inefficient.
d. Lisp uses dynamic type checking, resulting in some programming errors that are not detected until run-time.

28. Prolog is a logic programming language in which facts, rules, and goals can be expressed. Goals are achieved using a depth-first search strategy, proceeding through the facts and rules from top to bottom and left to right, backtracking if necessary. Below is a Prolog program with five facts and two rules.

```
parent (carol, john).
parent (john,mary).
parent (mary,jim).
parent (john,sue).
parent (david,jim).
ancestor (X,Y) :- parent (X,Y).
ancestor (X,Z) :- parent (X,Y), ancestor (Y,Z).
```

The following Prolog goal is given.

```
ancestor (john,X).
```

For this goal Prolog will deduce, in this order:

a. {X = mary} and {X = sue}.
b. {X = mary}, {X = jim}, and {X = david}.
c. {X = mary}, {X = sue}, and {X = jim}.
d. {X = mary}, {X = jim}, and {X = sue}.

29. An essential part of Prolog is the matching up of two expressions, resulting in a substitution that can make them equal. This process is called:

a. Evaluation.
b. Mechanical theorem-proving.
c. Resolution.
d. Unification.

30. Which of the following are true concerning the depth-first search strategy of Prolog?

a. This search strategy guarantees that infinite loops are avoided in looking for solutions.
b. This search strategy was chosen because it is space efficient.
c. This search strategy relieves the programmer of

concern with the order in which facts and rules are given.

d. This search strategy means that Prolog is not a pure logic programming language.

31. Interest in object-oriented programming has produced a new vocabulary. Match the following terms used in object-oriented programming with their closest meaning.

Term	Meaning
1. Class.	a. A collection of data and operations.
2. Message.	b. A description of a set of objects.
3. Method.	c. A procedure body.
4. Object.	d. A procedure call.

32. In object-oriented programming languages *inheritance* is a facility for defining a new class as an extension of previously defined classes. Which of the following are true concerning inheritance in most object-oriented programming languages, including C++ and Smalltalk?
    a. A class inherits variables from its superclass.
    b. Variables must be explicitly declared in the class that inherits them.
    c. A class may define new methods and override inherited methods.
    d. The search for a method proceeds from a class to its subclasses.

33. Object-oriented programming languages, including C++ and Smalltalk, allow the definition of functions that may be applied to different data types. The use of a single function for different types of data is called:
    a. Polymorphism.
    b. Instantiation.
    c. Message passing.
    d. Method search.

34. Of the following, the most important contribution of object-oriented programming is in providing an effective way to:
    a. Produce efficient code.
    b. Reuse code.
    c. Automatically generate code.
    d. Verify the correctness of code.

Syntax and Semantics

This final section has questions about Backus-Naur form (BNF) and semantics. BNF is a metalanguage used to describe the syntax of programming languages.

The following BNF grammar is used in questions 35–37. Nonterminal symbols in the grammar are **⟨s⟩**, **⟨t⟩**, and **⟨w⟩**. Terminal symbols are **@**, **!**, **#**, and **a**. The starting symbol is **⟨s⟩**.

```
⟨s⟩ ::= ⟨s⟩ @ ⟨t⟩ | ⟨t⟩
⟨t⟩ ::= ⟨w⟩ ! ⟨t⟩ | ⟨w⟩
⟨w⟩ ::= # ⟨w⟩ | a
```

The grammar describes a simple language of expressions with operators @, !, and #, and operand a. Let this grammar also reflect the precedence and the associativity of the operators.

35. List all sentences in the language (derived strings consisting only of terminal symbols) consisting of three or fewer symbols.

36. Give the precedence of the three operators, from highest to lowest.

37. Tell whether the @ and ! operators are left associative (consecutive operators of the same type are evaluated or grouped from left to right) or right associative.

38. There are certain areas of syntax that cannot be defined by a BNF grammar. Which of the following constructs of Pascal cannot be described by BNF?
    a. Correctly nested if statements.
    b. Procedure calls with the same number of parameters as the procedure declarations.
    c. Properly balanced and matched parentheses in arithmetic expressions.
    d. Variables declared the correct number of times inside procedures.

39. An important question about a BNF grammar for a programming language is whether the grammar is ambiguous. The existence of an algorithm to decide whether a BNF grammar is ambiguous is best described by:
    a. Such an algorithm exists.
    b. Such an algorithm exists, but is computationally intractable.
    c. Such an algorithm may exist, but has not yet been discovered.
    d. Such an algorithm cannot exist, because the question of determining the ambiguity of a BNF grammar is undecidable.

40. Most programming language manuals define the semantics of languages using informal descriptions and examples. Which of the following are reasons that formal methods for describing semantics are not commonly used in language manuals?
    a. Formal methods for describing semantics have not been extensively studied and are not well developed.
    b. The best known formal methods for describing semantics, operational, denotational, and axiomatic methods, are usually not sufficient to completely describe the semantics of programming languages.
    c. Formal methods for describing semantics are usually too complex to be of value to users of the language.
    d. Informal methods are usually completely satisfactory for describing the semantics of programming languages.

41. *Operational semantics* defines the effects of each language construct by the actions of an abstract machine. Which of the following are true concerning operational semantics?
    a. An actual implementation of the defined language must implement the structure of the abstract machine.
    b. An actual implementation of the defined lan-

guage must implement the effects of the abstract machine.

c. Operational semantics clearly distinguishes between the essence of a language concept and a specific implementation.

d. The efficiency of the abstract machine is an important issue.

Part II. Suggested Responses

1. d. [Pratt, p. 61]
2. c. [Pratt, pp. 87–89]
3. c. [Ghezzi, p. 119]
4. b. [MacLennan, p. 184]
5. a. [Pratt, p. 56]
6. a,d. [MacLennan, p. 204]
7. b,c. [Ghezzi, pp. 34–36, 58, 237]
8. a. [Pratt, pp. 286, 438]
9. a. [Pratt, p. 170]
10. c. [Sebesta, pp. 178–187]
11. 26, if * is evaluated before F(A), and 46, otherwise. [Sebesta, pp. 146–147]
12. b,c. [Sebesta, pp. 154–155]
13. 2^N. [Wilson, pp. 149–151]
14. a,b. [Pratt, pp. 185, 188–189]
15. 3, 5, 6. [Appleby, pp. 194–195]
16. d. [Wilson, pp. 229, 236]
17. b. [Sethi, pp. 359–360]
18. a,c. [Wilson, pp. 64–65]
19. 1) 5. 2) 7. [Sebesta, pp. 125, 130]
20. 1) B = 5 and C = 10. 2) B = 6 and C = 10. 3) B = 6 and C = 12. [Sebesta, pp. 264–270]
21. c. [Pratt, p. 287]
22. c,d. [Sethi, pp. 169, 173]
23. b. [Sethi, pp. 170–174]
24. d. [Wilson, p. 263]
25. a,b,d. [Sethi, pp. 254, 268]
26. b. [MacLennan, pp. 432–434]
27. c,d. [Sethi, pp. 254–255]
28. c. [MacLennan, pp. 488–489, 514–515]
29. d. [Kamin, pp. 376, 402]
30. b,d. [Ghezzi, pp. 306, 311]
31. 1-b, 2-d, 3-c, 4-a. [Sethi, p. 212]
32. a,c. [Sethi, pp. 219–221, 235–236]
33. a. [Kamin, p. 343]
34. b. [Kamin, p. 273]
35. a, a@a, a!a, #a, ##a. [Sebesta, pp. 76–77]
36. # has the highest precedence, ! has lower, and @ has the lowest precedence. [Sebesta, p. 80]
37. @ is left associative and ! is right associative. [Sebesta, pp. 82–83]
38. b,d. [Pratt, pp. 325–326]
39. d. [Pratt, p. 342]
40. c. [Pratt, pp. 345–348]
41. b. [Ghezzi, pp. 48, 317]

Part III. References

1. Appleby, D., Programming Languages: Paradigm and Practice, McGraw-Hill, New York, 1991.
2. Ghezzi, C., and Jazayeri, M., Programming Language Concepts, 2nd ed., Wiley, New York, 1987.
3. Kamin, S.N., Programming Languages: An Interpreter-Based Approach, Addison-Wesley, Reading, Mass., 1990.
4. MacLennan, B.J., Principles of Programming Languages: Design, Evaluation, and Implementation, 2nd ed., Holt, Rinehart and Winston, New York, 1987.
5. Pratt, T.W., Programming Languages: Design and Implementation, 2nd ed., Prentice-Hall, Englewood Cliffs, N.J., 1984.
6. Sebesta, R.W., Concepts of Programming Languages, Benjamin/Cummings, Redwood City, Cal., 1989.
7. Sethi, R., Programming Languages: Concepts and Constructs, Addison-Wesley, Reading, Mass., 1989.
8. Wilson, L.B., and Clark, R.G., Comparative Programing Languages, Addison-Wesley, Reading, Mass., 1988.

Additional References

9. Gelernter, D., and Jagannathan, S., Programming Linguistics, The MIT Press, Cambridge, Mass., 1990.
10. Marcotty, M., and Ledgard, H., Programming Language Landscape: Syntax, Semantics, and Implementation, 2nd ed., SRA, Chicago, 1986.
11. Sammet, J.E., Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs, N.J., 1969.
12. Tucker, A.B., Programming Languages, 2nd ed., McGraw-Hill, New York, 1986.

Epilogue

Now that you have reviewed this self-assessment procedure and have compared your responses to those suggested, you should ask yourself whether this has been a successful educational experience. The Committee suggests that you conclude that it has only if your have:

- discovered some concepts that you did not previously know about or understand, or
- increased your understanding of those concepts that are relevant to your work or valuable to you.

About the Author:
MITCHELL H. CLIFTON is an assistant professor of mathematics and computer science at West Georgia College. **Author's present address:** Department of Mathematics and Computer Science, West Georgia College, Carrollton, GA 30118.

Previous Self-Assessment Procedures

Self-Assessment Procedure I
Three Concept Categories within the Programming Skills and Techniques Area
May 1976

Self-Assessment Procedure II
System Organization and Control with Information Representation, Handling, and Manipulation
May 1977

Self-Assessment Procedure III
Internal Sorting
September 1977

Self-Assessment Procedure IV
Program Development Tools and Methods, Data Integrity, and File Organization and Processing
February 1978

Self-Assessment Procedure V
Database Systems
Peter Scheuermann and C. Robert Carlson
August 1978

Self-Assessment Procedure VI
Queueing Network Models of Computer Systems
J. W. Wong and G. Scott Graham
August 1979

Self-Assessment Procedure VII
Software Science
M. H. Halstead and Victor Schneider
August 1980

Self-Assessment Procedure VIII
The Programming Language Ada
Peter Wegner
October 1981

Self-Assessment Procedure IX
Ethics in Computing
Edited by Eric A. Weiss, from a book by Donn B. Parker
March 1982

Self-Assessment Procedure X
Software Project Management
Roger S. Gourd
December 1982

Self-Assessment Procedure XI
One Part of Early Computing History
Eric A. Weiss
July 1983

Self-Assessment Procedure XII
Computer Architecture
Robert I. Winner and Edward M. Carter
January 1984

Self-Assessment Procedure XIII
Binary Search Trees and B-Trees
Gopal K. Gupta
May 1984

Self-Assessment Procedure XIV
Legal Issues of Computing
Jane P. Devlin, William A Lowell, and Anne E. Alger
May 1985

Self-Assessment Procedure XV
File Processing
Martin K. Solomon and Riva Wenig Bickel
August 1986

Self-Assessment Procedure XVI
Computer Organization and Logic Design
Glen G. Langdon, Jr.
November 1986

Self-Assessment Procedure XVII
ACM
Eric A. Weiss
October 1987

Self-Assessment Procedure XVIII
Data Communications
John C. Munson
March 1988

Self-Assessment Procedure XIX
Copyright Law
Riva W. Bickel
April 1989

Self-Assessment Procedure XX
Operating Systems
J. Rosenberg, A. L. Ananda, and B. Srinivasan
February 1990

Self-Assessment Procedure XXI
Concurrency
Brian A. Rudolph
May 1990

Self-Assessment Procedure XXII
Ethics
Edited by Eric A. Weiss, from a report by Donn D. Parker, Susan Swope, and Bruce N. Baker
November 1990