This introduction to the principal features of Ada emphasizes those aspects expected to offer significant advantage in large-scale program development.

What is Ada?

Ronald F. Brender
Isaac R. Nassi
Digital Equipment Corporation

During the five years that Ada was being developed under the sponsorship of the US Department of Defense, considerable effort went into determining what requirements a language intended principally for embedded computer applications had to satisfy. The resulting language, however, is suitable not only for embedded computer applications, but also for general systems programming, real-time industrial applications, general applications programming, numeric computation, and for teaching good programming practices. This article introduces some of the concepts and features of Ada from which it derives its strength.

Modularization and large-scale software development

In recent years, many efforts have been made to reduce the complexity of large-scale software development projects. A number of these efforts are based on the notion of modularization—that is, partitioning solutions to large, complex problems into smaller, more understandable, and hence more manageable components.

Software maintenance was also a major concern in the design of Ada. Various kinds of inconsistencies that normally occur during the evolution of large systems will not occur in systems developed in Ada. A compilation data base allows modularization and separate compilation of components without giving up the ability to provide program-wide type checking.

Ada encourages, even demands, what has been called a "constructive" approach to programming. The facilities in Ada have been formulated to provide mechanisms for modularization.

The value of abstraction* in programming—long appreciated by programmers and language designers—has received much well-deserved publicity in recent years. In most languages, the procedure is the primary abstraction mechanism. To a lesser extent, separate compilation offers a means for modular grouping of related procedures and, perhaps, some common data objects. However, separate compilation as a means of abstraction is not directly supported in the widely used languages; when the concept has been employed, it has been through conventions for using the language. In Ada, the notion of abstraction has been elevated to a prominent position.

Packages. The concept of the Ada package is probably the language's principal contribution to the programming art. A package consists of two components—a specification part and an implementation part. Each is a unit, which means that each can reside in a separate file and can be compiled separately.

*We use the term "abstraction" in a general way to mean the process by which we distinguish the functional characteristics of a facility from the implementation of that facility.

0018-9162/81/0600-0017$00.75

The following example† illustrates the structure of a package:

```
(File A)
  package MY_PACKAGE is      the visible and private parts
  |                          form the package
  |                          specification
  |
  |                          this is the visible part
  |
  |
  private
  |
  |                          this is the private part
  |
  end MY_PACKAGE;

(File B)
  package body MY_PACKAGE is     this is the package
  |                              body
  |
  |
  |
  |
  end MY_PACKAGE;
```

The package specification is simply a sequence of declarations. The sequence up to the occurrence of the word "private" is called the visible part of the specification. The last part of a package specification is called the private part. The private part is not always needed and will be further described below.

The package body is the means by which the package provides the operations promised in the package specification. In the body, the code for the operations is given. The body may also contain other declarations needed to implement the operations, but they need not, indeed should not, be known or usable outside of the package body.

The package is thought of as a set of facilities provided for the benefit of other packages and procedures.

The package is thought of as a set of facilities provided for the benefit of other packages and procedures. The package specification is the means by which the names of these facilities are made available. The declared names identify the facilities of the package. These names can define operations in the form of procedures and functions. For operations, only the name, the type of the returned value (if any), and the names and types of the formal parameters are given. The names given in the specification can also define types, exceptions, tasks, and constant and variable data objects.

An example of a specification defining a simple package to perform I/O on a line-oriented text file is:

```
package TEXT is
  type FILE is private;
  NO_FILE : constant FILE;
  type MODE is (READ, WRITE);
  IO_ERROR : exception;
  procedure OPEN
    (F : out FILE;  M : in MODE;  NAME : in STRING);
  procedure GET (F : in FILE;  S : out STRING);
  procedure PUT (F : in FILE;  S : in STRING);
  procedure CLOSE (F : in FILE);
private
  type FILE is new INTEGER;
  NO_FILE : constant FILE := 0;
end TEXT;
```

†Vertical bars are used in this and later examples to indicate the possible presence of declarations or statements not relevant to the example.

In this example, the type FILE is introduced and used in the visible part of the specification, but its actual definition does not appear until the private part. Users of the package can declare variables of type FILE, but they are not able to exploit intentionally or accidentally the fact that a FILE variable is represented as an integer. Presumably, the package body for this example would use the FILE value as an index into an internally maintained table of more traditional file descriptors.

NO_FILE is a variable that is constant—that is, its value cannot be changed even in the package body. It represents a file that is not opened. Its representation as the integer value zero is also private. Type MODE is defined as an enumeration—that is, as an ordered set of discrete, named literal values. Because the complete definition is given in the visible part, all of the operations for enumerations are available to a user.

The procedures introduce the operations that can be used. Note that the bodies of the procedures are not given; these would be given in the package body (not shown).

Abstraction. The key to abstraction is information hiding. Ada provides two principal mechanisms for hiding details of implementation while exporting, or making available to users, the mechanisms for using the implementation. The first has already been described. The fact that the package is composed of two physically separate parts—the specification and the body—forms the basis for hiding the implementation details of the package. Indeed, given the specification, it is possible to compile programs which make use of the package before the package body even exists. All of the information needed to compile programs referencing a package is given in the package specification.

The second mechanism is the private part of the package specification. Its purpose is to provide a place for declarations that are necessary to define the *physical* interface of the package, but which must not be included in its *logical* interface. The physical interface of a package provides the information that a compiler must have when compiling a program that uses facilities provided by the package. The logical interface is simply the visible part of the package. The distinction between physical and logical interface is not meant to imply that the information contained in the private part is invisible to the user of a package. It is not. Instead, it implies that the user of the package is unable to make use of the information contained in the private part in any way that will affect the correctness of his program.

In the example specification for the package TEXT, the declaration of the private type FILE following the word "private" is part of the physical interface. This allows compilers to allocate storage for file objects. This distinction between physical and logical interfaces permits the following maintainability characteristics to hold.

- Changes to the package body are guaranteed, first of all, not to require modifications to the source of programs referencing the package and, secondly, not to require recompilations of those programs.
- Changes to the private part are guaranteed not to require any modifications in the source code of programs that reference the package, but may require recompilation of the source.
- Changes to the visible part may require changes to the source code, and hence recompilation. For example, the number of parameters of a procedure might be changed.

Control over the name space. One of the most difficult problems faced by large programming projects is the management of names and the sharing of declarations to be used by many programs. This problem manifests itself in large, relatively unstructured, and frequently difficult to manage files of shared declarations; complex naming conventions; and/or multiple declarations of symbols. Algol-like scope rules have been considerably extended to form the basis of name management in Ada.

Ada facilitates management of a program's name space by requiring each compilation unit to indicate the other compilation units with which the unit is to be compiled and upon which it depends. For example, package C might begin with the clause "with A, B;" which states that package C can refer to the names A and B, which must be available in the compilation library, and can use names declared in A and B. Even though A or B might depend on other compilation units, this is not reflected in C—unless, of course, C needs to use these other units directly.

Name qualification aids in keeping symbols distinct. For example, consider the following unit which makes use of the package TEXT illustrated earlier:

```
with TEXT;
procedure COPY_TEXT (IN_FILE_NAME,
  OUT_FILE_NAME : STRING) is
  MAX_STRING_LENGTH : constant := 133;
  IN_FILE, OUT_FILE : TEXT.FILE := TEXT.NO_FILE;
  LINE : STRING (1..MAX_STRING_LENGTH);
begin
  TEXT.OPEN (IN_FILE, TEXT.READ,
    IN_FILE_NAME);
  TEXT.OPEN (OUT_FILE, TEXT.WRITE,
    OUT_FILE_NAME);
  loop
    TEXT.GET (IN_FILE, LINE);
    TEXT.PUT (OUT_FILE, LINE);
  end loop;
exception
  when TEXT.IO_ERROR =>
    TEXT.CLOSE (IN_FILE);
    TEXT.CLOSE (OUT_FILE);
  when others =>
    raise;
end COPY_TEXT;
```

By default, a name declared in a package specification must be denoted outside the package using the name of the package and a dot as a prefix. Two examples above would be TEXT.FILE and TEXT.OPEN.

As a programming convenience, a "use clause" can be written to indicate that names in a package can be used without the package name prefix. A use clause is shown in the example below, where only selected parts of the example given above are repeated.

```
with TEXT;  use TEXT;
procedure COPY_TEXT (. . .) is
  . . .
  IN_FILE, OUT_FILE : FILE := NO_FILE;
  . . .
begin
  OPEN (. . .);
  OPEN ( . . .);
  . . .
end COPY_TEXT;
```

Ada's rules assure that confusion cannot result when, for example, there are declarations of the name FILE in two or more packages that are used together in the same unit. In such a case, the compiler will detect and report the ambiguity. The program can then be changed to use qualified names as needed to avoid the ambiguity.

Large files incorporated by text inclusion are avoided, because a programmer specifies in a with clause only those packages whose constituents are referenced. Admittedly, the rules for determining the possible meanings of a name are more complicated than in familiar languages. However, the problem the rules are trying to solve is itself complicated.

Overloaded names. In Ada, names of subprograms can be overloaded—that is, the same name can be used for more than one subprogram (even in the same scope), provided the different declarations can be distinguished by the number, types, and names of the parameters and the result type. For example,

```
declare
  procedure P (X : INTEGER) is . . . ;
  procedure P (X : FLOAT) is . . . ;
  A : INTEGER;
  B : FLOAT;
begin
  P (A);     calls the first declared P
  P (B);     calls the second declared P
end;
```

This facility contributes to both abstraction and name-space management. Abstraction is aided because the same procedure name can be used for conceptually equivalent operations on different types of data. For example, SQRT can be used for the square root operation for the various precisions of floating-point types. Name

management is aided because fewer names are needed, and naming conventions can be simplified or avoided.

Overloading can also be applied to the predefined operations of Ada so that they can be used with user-defined types. For example, a package that defines complex arithmetic, which is not part of standard Ada, would undoubtedly have a specification resembling the following example.

```
package COMPLEX_ARITHMETIC is
  type COMPLEX is
    record
      RE : FLOAT;
      IM : FLOAT;
    end record;
  function " + " (X, Y : COMPLEX) return COMPLEX;
  function " - " (X, Y : COMPLEX) return COMPLEX;
  |    other functions or procedures for type COMPLEX
  |
end COMPLEX_ARITHMETIC;
```

In this example, the predefined operators + and − are overloaded so that they can be used with operands of the user-defined type COMPLEX.

Overloading can be applied to the predefined operations of Ada so that they can be used with user-defined types.

While overloaded names make the compiler's job harder—that is, it has to determine which declaration is the right one to use—they nevertheless make it easier to write understandable programs.

The compilation data base. Units of Ada source text can be separately compiled. Traditionally the problem with separately compiled sources has been that they offer an opportunity to inadvertently breach the type system. For example, a procedure defined with several parameters in one source file is called with too few parameters or with parameters in the wrong order in another source file.

The Ada viewpoint is that compilation of several units can be separate (that is, individual units can be compiled without compiling the complete program) but not independent (that is, in the absence of information about other parts of the program). Compilations are done in the context of a data base of previously compiled units. The Ada compiler manages the data base.

The language is defined so that the compiler can recognize and keep track of the logical dependencies between units in separate sources. If unit B makes use of information defined in unit A, then, naturally, unit A must be compiled before unit B. Moreover, if unit A is modified and recompiled, then the compiler knows that unit B, as well as any units that depend on B, is no longer valid and must eventually be recompiled as well. The compiler will then refuse to compile any unit that depends on B until B itself is recompiled.

An Ada compiler or another component of an Ada programming system is intended to provide means for querying the recompilation status of the data base, although this is not part of the language definition. Furthermore, before any unit is incorporated into an executable program, the data base would be checked to assure that no invalid units are required.

Naturally, certain kinds of program modification require that other dependent units be recompiled. However, the Ada data base is a source of information that will help in minimizing the amount of recompilation while preserving the same degree of consistency checking across separately compiled units that is possible within a single unit.

Methods of development. Ada directly supports both bottom-up and top-down software development methodologies.

Bottom-up development can proceed as follows. A member of the programming team is selected to be responsible for a package. The team agrees on a package specification. Subsequently the team need discuss the package only for proposed changes to the specification. The person responsible for the package compiles a specification for the package. The compiler automatically inserts it in the data base. Other team members then compile other units which refer to that specification. Sometime before the package is incorporated into an executable program, the person responsible for the package writes and compiles the body.

Top-down development is accomplished with the aid of program stubs and subunits. A stub indicates the place where a separately compiled subunit must eventually be provided. For example, while developing package P, a programmer might write:

```
(File P0)
  package body P is
  |
  |
    type MY_TYPE is array (1. .10) of INTEGER;
    procedure NOT_YET
      (X : MY_TYPE) is separate;      stub
  |
  |
  begin
  |
  |
  |
  end P;
```

The reserved word "separate" in this example tells the compiler that a subunit consisting of the procedure named NOT_YET with one parameter of type MY_TYPE will be compiled later. The compiler saves in the data base all of the contextual information needed from P for the subsequent compilation of NOT_YET.

Later, a programmer can compile the following subunit.

```
(File P1)
  separate (P)
  procedure NOT_YET (X : MY_TYPE) is
  |
  |
  begin
  |
  |
  end;
```

In this case, the key word "separate" and its argument P tells the compiler that what follows is a subunit of P and must be compiled as though it were completely declared where indicated in unit P. The context surrounding the declaration of NOT_YET in P is automatically recalled to begin the compilation. Thus, references to items declared in P, such as MY_TYPE, are allowed within NOT_YET. In this way, top-down programming can proceed in an orderly, controlled manner.

Systems programming in Ada

Although systems programming has been characterized in a number of different ways, the following features must be present in any language intended to be a systems programming language.

- It must be possible to generate very efficient code for the language.
- It must be possible to describe interfaces to programs written in other languages, including assembly language, as well as to describe and access their data structures efficiently. These data structures are likely to be arbitrarily complex, and the existing programs are likely to have complicated calling sequences and conventions.
- It must be possible to access various components of the instruction set architecture, such as fixed locations in memory, and to directly interface to hardware interrupts.
- The language must facilitate the construction of very reliable programs.

Ada has these characteristics.

Efficiency. Designing the language has involved particular attention to assuring that efficient data access and code efficiency are possible.

The language supports a full complement of predefined data types—such as integer, fixed and floating-point numbers, pointers, and characters—as well as arrays of arbitrary dimension (strings are defined as arrays of characters) and records (which are collections of data objects that need not have the same type). These data structures can be composed in arbitrarily complex ways, permitting arrays of records, records embedded within records, arrays within records, and so on. Subarrays—that is, slices along a dimension of an array—are provided for arrays of one dimension (i.e., vectors) so that an array or subarray is always a contiguous region of storage.

The type system is constructed so that implicit run-time description of data is not required except in traditional cases, such as descriptors that define the bounds of arrays when their bounds are not known during compilation, and discriminant constraints that select from among variant components of records. While there are some exceptions to this general statement, discussing them would be going beyond the scope of this article.

Parameter passing in Ada is unique. The language is intentionally left sufficiently vague to allow compiler implementations to choose the parameter passing mechanisms best suited to the objects being passed, while still maintaining a meaning very close to that of copy semantics. Each parameter has a parameter mode of "in," "out," or "in out" indicating how the parameter will be used. (There is no analog to the "var" versus "val" modes found in Pascal, for example.) In practice, nearly all parameters of array, record, or private types can be passed by reference, although for casual purposes, it is generally acceptable to understand the process as one in which actual parameters are copied into the data space of the called procedure. Scalar and access—that is, pointer—actual parameters are always passed using copy semantics.

Code generation is Ada is similar to that in other procedural languages such as Fortran, Pascal, or PL/I. There are no language-specific obstacles to generating efficient code for a variety of conventional machine architectures.

Interfacing to existing environments. The language segregates and controls references to non-Ada environments in order to facilitate program transportability.

Representation specifications allow the programmer to specify the exact mapping between the bits of a data object and the underlying storage in a straightforward way. Thus, it is possible to describe the structure of machine registers or hardware-defined data structures. Calling routines written in other languages or invoking system services while trying to cooperate with a host operating system is also straightforward, although nonstandard calling sequences may require compiler support.

Ada directly supports both bottom-up and top-down software development methodologies.

"Pragmas" are constructs that generally provide compiler directives to influence the compilation without affecting the meaning of the program. One of these, the interface pragma, indicates that a procedure specification corresponds to a procedure written in another language. In conjunction with this, overloaded procedure names allow a user to declare a procedure written in another language that can take variable numbers and types of arguments.

Finally, there are predefined subprograms that provide an escape from the type system. In particular, these allow a programmer to view an object of one type as if it were of any other type. For example, a floating-point value could be viewed in terms of its hardware representation as a packed record with several fields which are arrays of bits. These operations are totally run-time efficient in that they generate no code.

Reliability. The language is strongly typed in the sense that all type-checking is done at compile-time. However, certain forms of run-time type parameterization, such as dynamic range constraints, are possible. The checking of these parameters occurs during execution. If maximal run-time performance is desired, these checks can be

suppressed by using a pragma—with the obvious loss of safety.

When constraint violations are detected, exceptions are raised. Exception handlers are highly structured, and programmers can easily write their own. Exceptions are of two varieties, predefined and user-defined. Both are handled in precisely the same way since the language does not distinguish between the two with special rules. When an exception is raised, execution of a handler replaces the exception of the block or subprogram body in whose dynamic context the exception occurred.

Concurrency

For real-time applications, Ada provides facilities for multitasking—that is, for logically parallel threads of execution that can cooperate in carefully controlled ways.

Tasks. A task is an independent thread of execution. Like a package, a task is divided into a specification part and a body. Likewise, the modularity and abstraction concepts discussed for packages generally apply to tasks as well. The task specification contains entry declarations that define the procedure-like calls that can be made to communicate with the task. The task body contains the code and variables—that is, the internal state—defining the behavior of the task.

Below is an example of a task to provide an asynchronous buffer between a line-oriented producer and a character-oriented consumer. This task declaration is assumed to be in the context of declarations of the types LINE and CHARACTER; the latter is, in fact, a predefined type.

```
task LINE_TO_CHAR is        this is the task specification
   entry PUT_LINE (L : in LINE);
   entry GET_CHAR (C : out CHARACTER);
end LINE_TO_CHAR;

task body LINE_TO_CHAR is       this is the task body
   BUFFER : LINE;
begin
   loop
      accept PUT_LINE (L : in LINE) do      accept statement
         BUFFER := L;                         and its body
      end PUT_LINE;
      for I in BUFFER'FIRST . . BUFFER'LAST loop
         accept GET_CHAR (C : out CHARACTER) do
            C := BUFFER (I);
         end GET_CHAR;
      end loop;
   end loop;
end;
```

In this example, the task specification declares the entries PUT_LINE and GET_CHAR for use by other tasks. The body declares a local variable BUFFER which is used to hold a complete line until all of its characters have been transmitted.

The code for the task consists of an unbounded loop with two basic parts. The first part is an accept statement for entry PUT_LINE. The accept statement looks quite similar to a procedure declaration; it has formal parameters and a body. The task waits at this point until some other task calls the entry PUT_LINE to provide data for the buffer. The second part is another loop that transmits one character at a time.

The example above illustrates the declaration of a single task object. Task types can also be declared, and then any number of tasks of that type—that is, all with the same properties—can be declared as objects. Indeed, because tasks are objects, they can be components of records or arrays. They can be pointed to by access objects, passed as parameters, created dynamically, and so on. However, task "values" cannot be assigned.

Intertask communication. When an entry has been called and the called task reaches an accept statement for that entry, a "rendezvous" is said to occur. The two tasks "come together," and only a single thread of execution is active for the duration of the accept statement. You can think of this single thread of execution as belonging to either or both of the two tasks. At the end of the accept statement, the rendezvous is complete. Both tasks then continue independently and asynchronously.

In the last example given, when the task is waiting at PUT_LINE, a call on GET_CHAR is queued and the caller must wait. Conversely, when the task is waiting at GET_CHAR, a call to PUT_LINE is queued. Any number of tasks can be waiting for a given entry to be accepted. In this way, coordination is assured between tasks that provide lines and those that consume characters.

While the last example did not require more than one accept statement, there can be any number of accept statements for each declared entry. Other features allow accept statements to be conditionally executed.

The example below illustrates more complicated waiting for entry calls.

```
select
   accept ENTRY_1 ( . . . ) do
   |
   end;
or
   when SOME_CONDITION = >
      accept ENTRY_2 ( . . . ) do
      |
      end;
or
   |
   |     other accept alternatives
   |
else
   |
   |     "else" alternative
   |
end select;
```

A select statement allows a task to wait until a call is received on any one of a set of entries. If more than one call is waiting when the select starts, one of them will be arbitrarily chosen for processing.

The set of possibilities need not be the same on every execution of the select statement. An optional "when clause"—such as shown preceding accept ENTRY_2 in

the example above—must evaluate to true for the entry to be "open," that is, eligible to be accepted. If none of the alternatives are open and the optional else alternative appears in the select statement, it will be chosen. An exception is raised if no alternative is open and no else is provided.

Other variations of the select statement include a delay or terminate alternative which can appear in place of an accept statement. However, neither a delay nor terminate alternative can be used in combination with the else alternative. If one of the alternatives of the select is a delay and there are no calls waiting for any of the open alternatives, the task at most will wait the specified amount of time before proceeding. A delay value of zero means that if no entry call is immediately available, the task will not wait at all.

A terminate alternative is a particularly interesting construct that addresses the question, "How do you gracefully shut down a set of cooperating tasks without running the risk of error?" When a task is waiting at a terminate alternative, it effectively declares to the run-time environment that it is prepared to be terminated provided that all other tasks with which it can interact directly or indirectly are also at terminate alternatives or have completed. When all such tasks are in this state, none of the tasks of the set will ever receive an entry call (they are all waiting for some other task to make an entry call), so it is safe to terminate them all. Of course, if an entry call is received by a task that is prepared to terminate, the "okay-to-terminate-me" status is rescinded and normal processing resumes.

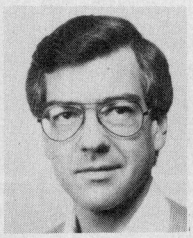
The priority of the external device is dependent on its hardware priority and is guaranteed to be higher than any software priority.

The language also supports a conditional entry call in which the calling task is not suspended and the entry call is not made unless the called task is ready to immediately accept it. There is also a timed entry call—a generalized form of the conditional entry call—in which the call is cancelled if it is not accepted within a specified amount of time. These two forms of entry call make it possible for a calling task to assure that is will not be indefinitely delayed by any task that it calls.

Priorities and scheduling. A task can either have or not have a priority. A priority is specified by a pragma. If a priority is given, it must be a compile-time static value and cannot be changed during execution. When no priority is given, the compiler is free to choose when scheduling decisions need to be made, when the overhead of the scheduling decision can be avoided, and at what priority the task is run when it starts executing. The overhead of tasking in Ada is small. When implemented on a single processor, it is comparable to the overhead of procedure calls.

Interrupts. In Ada, external devices are treated as tasks, and interrupts are treated as entry calls. The priority of the external device is dependent on its hardware priority and is guaranteed to be higher than any software priority. In this way, interrupts can be directly "connected" to accept statements. A consequence is that the tasking model is powerful enough to incorporate the conventional view of interrupts and interrupt handlers.

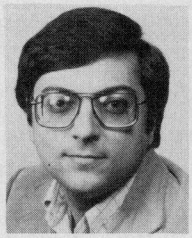
We have touched on a number of the features of the Ada language, pointing out some of the concepts employed. The language draws on many years of research into algorithmic languages and programming methodology. It incorporates and directly supports modern programming concepts of abstraction and modularization, separate compilation of program units without loss of program-wide checking, concurrency, and features for efficient systems programming. Furthermore, Ada will be supported and widely available because of its support by both the Department of Defense and many commercial hardware and software vendors. Ada is certain to affect the industry in a significant and beneficial way. ■



Ronald F. Brender joined Digital Equipment Corporation in 1970, where he has worked in both research and product development contexts. Working with researchers at Carnegie-Mellon University, he introduced the use of the original Bliss-10 to Digital. He headed development of Fortran IV-Plus (the first optimizing compiler implemented on the PDP-11) using Bliss-11. From 1973 to 1978, he represented Digital on the ANS X3J3 Fortran Standards Committee. He formed and led the development team that designed and produced the Bliss-16, Bliss-32, and Bliss-36 language and compilers, and promoted their use in transportable and nontransportable applications.

In 1978, Brender was named Digital fellow. He spent a year as a visiting scientist in the CMU Computer Science Department where he did research in data abstraction languages and the formal semantics of programming languages. Since his return to Digital's Corporate Research Group, he has been working with Ada. His interests include the design and implementation of high-level languages, especially for software development, and software development methodology. He is a member of the IEEE Computer Society, ACM, SIGPLAN, and SIGOPS.

Brender received the BSE in engineering sciences, the MS in applied mathematics, and the PhD in computer and communication sciences from the University of Michigan in 1965, 1968, and 1969, respectively.



Isaac R. Nassi is a consulting engineer and manager of the programming languages and software engineering section of Digital Equipment Corporation's Corporate Research Group. His principal research interests are programming languages and systems. He was involved in the development of production compilers for Bliss and Jovial. He became active in the Ada program in 1977 while commenting on the phase I language designs. As a consultant to the US Army, he helped to prepare the Army's comments on the phase II language designs. He was appointed as one of the original Ada Distinguished Reviewers by DARPA in 1979, at the start of phase III, and has since worked closely with the Ada language design team.

A member of the IEEE Computer Society and ACM, Nassi received his PhD from the State University of New York at Stony Brook.