THE RAND-ABEL™ PROGRAMMING LANGUAGE:
Reference Manual

Norman Z. Shapiro, H. Edward Hall,
Robert H. Anderson, Mark LaCasse

October 1985

N-2367-NA

Prepared for

The Director of Net Assessment,
Office of the Secretary of Defense

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited
A RAND NOTE

THE RAND-ABEL™ PROGRAMMING LANGUAGE:
Reference Manual

Norman Z. Shapiro, H. Edward Hall,
Robert H. Anderson, Mark LaCasse

October 1985

N-2367-NA

Prepared for

The Director of Net Assessment,
Office of the Secretary of Defense

---

知识产权

Accession For

NTIS CRA&I
DTIC TAB
Unannounced
Justification

By
Distribution:

Availability Codes
Dist
Avail a d/jor
Special

A-1

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED
**Title:** The Rand-Abel Programming Language: Reference Manual

**Authors:** Norman Z. Shapiro, H. Edward Hall, Robert H. Anderson, Mark LaCasse

**Performing Organization:** The Rand Corporation

**Address:** 1700 Main Street, Santa Monica, CA 90406

**Date:** October 1985

**Number of Pages:** 113

**Distribution Statement:** Approved for Public Release: Distribution Unlimited

**Keywords:** Programming Languages, War Games

**Abstract:**

See reverse side
This Note describes the RAND-ABEL programming language, which was designed to be suitable for (1) large rule-based systems, (2) war gaming and multiscenario sensitivity analysis, and (3) use by any of several governmental gaming and analysis organizations. RAND-ABEL is a preprocessor for the C programming language under the UNIX operating system, which makes RAND-ABEL quite portable across different computers. It is very fast in execution time when compared with other languages of similar readability. The RAND-ABEL language provides a number of unique capabilities. For example, it supports tables within the source code, for use as decision tables or to govern an iterative execution, and it is a strongly typed language, permitting certain kinds of errors in complex programs to be uncovered early. A related Rand publication, R-3274-NA, traces the evolution of RAND-ABEL, and discusses some of the requirements and principles underlying its design.
The RAND-ABEL\textsuperscript{TM} programming language\textsuperscript{1} was developed at The Rand Corporation for use in writing complex "agents" as part of a system for automated war gaming. The language was designed and implemented by Norman Z. Shapiro, H. Edward Hall, and Mark LaCasse for use by the Rand Strategy Assessment Center (RSAC). The RSAC and the work reported here are supported by the Director of Net Assessment in the Office of the Secretary of Defense under Contract MDA-903-85-C-0030.

RAND-ABEL is an operational language that is continuing to evolve. This Note documents the RAND-ABEL language as it exists operationally in September 1985. A subsequent version of the language will be operational within a few months, and will be documented thereafter. That version will include interpretive features, sets, and a substantially expanded capability for executable tables within the source code.

The present Note is intended for programmers and applications specialists that will be writing in the RAND-ABEL language. It gives a somewhat terse but complete description of the language. It assumes that the reader is fluent in at least one high-level programming language (or better yet, several) and is familiar with the notation and concepts used to describe the formal syntax of programming languages. Others will find this reference manual useful after becoming familiar with RAND-ABEL by other means. The following document traces the evolution of RAND-ABEL, and discusses some of the requirements and principles underlying its design:


\textsuperscript{1}RAND-ABEL is a trademark of The Rand Corporation.
Inquiries and comments on this Note are welcome. They may be sent directly to the authors or to Paul K. Davis, director of the Rand Strategy Assessment Center.
SUMMARY

This reference manual describes the RAND-ABEL programming language. In designing the RAND-ABEL language, we determined that

- RAND-ABEL should be suitable for large, rule-based systems.
  - It should lend itself to program development by multimember programming teams
  - It should be fairly easy to maintain

- RAND-ABEL should be suitable for war gaming and multiscenario sensitivity analysis.
  - Domain-substantive RAND-ABEL rules should be readable by domain specialists who are not RAND-ABEL programmers, and the code should be self-documenting
  - RAND-ABEL should be efficient in execution

- RAND-ABEL should be suitable for use by any of several governmental gaming and analysis organizations,
  - It should be transportable to various computers capable of hosting the UNIX operating system.

The RAND-ABEL language was designed for the specific requirements of the Rand Strategy Assessment Center (RSAC). RSAC is building a large system for automated and semi-automated war gaming in which separate models can represent U.S., Soviet, and third country behavior [Davis and Winnefeld, 1983; Shukiar, 1985]. RAND-ABEL is a preprocessor for the C programming language under the UNIX operating system, which makes RAND-ABEL quite portable across different computers. RAND-ABEL is very fast in execution time when compared with other languages of similar readability. We estimate that C language programs execute no more than three times faster than comparable ABEL programs.

In the RSAC environment, RAND-ABEL is used with a data dictionary, a data editor, and support for co-routines. This allows a flexible, hierarchical modeling system, allowing human teams to replace some of the models. Although designed for the RSAC, we anticipate that
RAND-ABEL will be of interest for other applications on UNIX systems requiring a highly readable language, fast performance, and early discovery of errors (for example, in the design of large rule-based models and simulations).

The RAND-ABEL language provides a number of unique capabilities. For example, RAND-ABEL supports tables within the source code, for use as decision tables or to govern an iterative execution. We find that table statements provide a much more succinct and readable alternative to long sequences of sentence-like rules typical of rule-based languages. When used as decision tables, RAND-ABEL tables correspond closely to the decision trees analysts and reviewers use in working out a logically complete argument. RAND-ABEL tables have a syntax that is inherently two-dimensional.

RAND-ABEL is a strongly typed language, permitting certain types of errors in complex programs to be uncovered early. Many of RAND-ABEL's features are derived from constructs in the C programming language.
Jean LaCasse tested the RAND-ABEL Translator's robustness and
diagnostics at various stages, helped debug some of the code, and wrote
preliminary documentation.

Several other members of the RSAC staff have made valuable
suggestions regarding RAND-ABEL syntax and requisite features. These
include Steven Bankes, Arthur Bullock, Paul Davis, William Jones,
Christe McMenomy, Ross Quinlan, and Herbert Shukiar.

We especially wish to acknowledge strong and continuing support for
the development of the RAND-ABEL language by Paul Davis and Herbert
Shukiar of the Rand Strategy Assessment Center (RSAC). Paul's stubborn
refusal to be satisfied with anything less than our best effort at
meeting RSAC's real needs, Herb's effective and knowledgeable guidance,
and their allocation of resources to RAND-ABEL when its future was
uncharted are responsible for RAND-ABEL's success to date.
CONTENTS

PREFACE ............................................................... iii
SUMMARY ............................................................... v
ACKNOWLEDGMENTS ........................................................... vii

Section
I. INTRODUCTION ................................................... 1
   Overview ..................................................... 1
   Notational Conventions ...................................... 4

II. NAMES, IDENTIFIERS, WHITE SPACE, AND COMMENTS .......... 7
   Names and Identifiers ........................................ 7
   A Note on White Space and Comments ....................... 8

III. DATA TYPES ..................................................... 11
    Primitive Data Types ........................................ 11
    Enumerated Data Types ....................................... 13

IV. VALUES, EXPRESSIONS, AND OPERATORS ............................ 15
    Values and Simple Expressions ............................... 15
    Operators ................................................... 21

V. DECLARATIONS .................................................. 28
    To Declare a Variable ....................................... 28
    To Declare an Array; MAKE and ERASE Statements for:
    Arrays .............................................................. 29
    To Declare a Function .......................................... 34

VI. FUNCTIONS .................................................... 36
    Defining a Function .......................................... 36
    Named Function Calls and Function Invocations ............ 37

VII. RAND-ABEL STATEMENTS ........................................... 40
    Assignment .................................................. 40
    Conditional Execution ....................................... 42
    Repetitive Execution ....................................... 43
    Table Statement ............................................... 46
    Functions: Invoking and Exiting ............................. 55
    Input/Output ................................................ 57
    Compound and Null Statements ................................ 64
I. INTRODUCTION

OVERVIEW

RAND-ABEL is a computer programming language implemented on the UNIX\(^1\) operating system. A program called the "RAND-ABEL Translator" compiles RAND-ABEL statements into a C program [Kernighan and Richie, 1978], which is in turn compiled and run.

RAND-ABEL was developed for use in the Rand Strategy Assessment Center (RSAC), to be used in the development of complex models. Six primary design goals guided the development of RAND-ABEL. RAND-ABEL is intended to be:

- Reasonably self-documenting. The RAND-ABEL code, by itself, should convey the meaning of a program.
- Understandable by English speakers familiar with the subject matter. Readers of the program should not need detailed programming knowledge to comprehend the program.
- Reasonably easy to learn and use by individuals with good analytic capability and modest programming skills. Analysts and application specialists with only some prior experience in a high-level programming language, such as FORTRAN, should be able to program effectively in RAND-ABEL without extensive training and study.
- Rapid in execution. Because RAND-ABEL was specially designed for building rule-based programs with many qualitative variables, it is important that these large programs be able to execute rapidly and efficiently.
- Portable across different types of computer hardware. RAND-ABEL and systems developed in it should not be unique to a single computer or a manufacturer's computers, but rather be portable across a range of minicomputers and powerful microcomputers.

\(^1\)UNIX is a trademark of AT&T Bell Laboratories.
Supportive of specialized RSAC needs, such as co-routines and tabular data, and well-suited to the creation of complex simulations by groups of developers.

Although the current version of RAND-ABEL is easy to learn for the simple programs most people work with, it requires a professional programmer to learn the intricacies of RAND-ABEL required to run the full RSAC simulation with all of its multiple coprocesses and interfaces. In its present form, RAND-ABEL is not as easy to use as we had expected. This shortcoming should be reduced with the imminent arrival of interpretive RAND-ABEL.

Prior to the development of RAND-ABEL, the Rand Strategy Assessment Center used the ROSIE language [Fain et al., 1981] for programming the Scenario Agent. The ROSIE program representing the Scenario Agent of the Mark II RSAC system is documented in Schwabe and Jamison, 1982. We found that RSAC programs executed in ROSIE orders of magnitude too slow for our future needs, which included operating large-scale simulations in a matter of minutes. Also, we sought special features (e.g., decision tables) that were not likely to be available in ROSIE. Therefore, RAND-ABEL was developed as a separate language. Nevertheless, much of the form and style of the statements in RAND-ABEL derive from its ROSIE heritage. It was found that analysts were using only a portion of the features of ROSIE, so that RAND-ABEL is a simpler language.

The goals of speed in execution and portability were achieved by writing a RAND-ABEL compiler (called the RAND-ABEL Translator) that translates RAND-ABEL statements into statements in the C language. Since C and its host operating system, UNIX, are available on a wide selection of computers, RAND-ABEL becomes similarly portable. In addition, efficient C language compilers are available, thereby permitting the efficient compilation of RAND-ABEL statements through this two-step compilation process. As a result, RAND-ABEL is currently available on those computers running the UNIX 4.2bsd operating system.

2The history, rationale, and design of RAND-ABEL is described in Shapiro et al., 1985.
and having a C language compiler. Moving RAND-ABEL to later versions of AT&T's UNIX system should be trivial.

RAND-ABEL is a strongly typed language; that is, the properties of all identifiers are declared before they are used. The RAND-ABEL compiler then uses these properties to test the validity of RAND-ABEL statements, so that certain types of errors (particularly control-related and data-related errors) may be caught at the earliest possible time.

The RAND-ABEL language contains a number of novel features. Perhaps the most novel construction in RAND-ABEL is the table statement; the following is valid RAND-ABEL code. (In this and other examples in this manual, RAND-ABEL keywords are printed in **boldface** to distinguish them from identifiers, constants, and comments chosen by the user to describe a particular application. Comments appear within square brackets.)

**Table** Deploy

[This table initiates the deployment of assigned forces]

<table>
<thead>
<tr>
<th>qty</th>
<th>#-%</th>
<th>unit-type</th>
<th>unit-owner</th>
<th>to-area</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>Denmark</td>
<td>CEur-1</td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>Netherlands</td>
<td>CEur-2</td>
</tr>
<tr>
<td>25</td>
<td>%</td>
<td>Troops</td>
<td>FRG</td>
<td>CEur-3</td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>UK</td>
<td>CEur-4</td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>Belgium</td>
<td>CEur-5</td>
</tr>
</tbody>
</table>

All RAND-ABEL tables consist of column headings followed by rows containing data. In this example, there are five columns. The meaning of the top row of the table, as defined by the Deploy function, orders 100 percent of the troops "owned by" Denmark to deploy to axis 1 in CEur (the Central European theatre). The table calls the Deploy function five times (once for each row of data) with five parameters (bound to the entry in each column of the table).

The table statement is a powerful device, capable of both defining iterative processes and creating decision tables. Its syntax is fully two-dimensional. A function call or RAND-ABEL statement (possibly a compound statement) occurring immediately after the table keyword is called once for each row in the table, with the table's column headings
parsed and matched with function parameters or variables in the statement. The table statement was developed because tables of information were found to be a natural means of representation for the strategic analysts who are the primary users of the language.

A second noteworthy feature of RAND-ABEL is "declaration by example." All identifiers are declared by giving examples of their use, usually by an assignment statement:

**Declare** message by example: *Let message be "I have Checkmate".*

In this manner, the data type associated with an identifier is declared without requiring the use of a whole vocabulary (e.g., integer, real, character string, boolean, process, enumerated variable, array) that may not be meaningful to analysts who are not professional programmers. Furthermore, it is especially useful in rule-based systems with many ad hoc data types that otherwise would require names for strong typing (i.e., the data types to which enumerated variables belong).

RAND-ABEL also has a built-in set of functions to handle co-routines, and a "data dictionary" to coordinate external data references among program modules being developed independently. These language features are discussed in Sections IX and X of this Note.

**NOTATIONAL CONVENTIONS**

This Note presents the form and content of the RAND-ABEL programming language. In doing so, it must use a set of stylistic conventions to represent RAND-ABEL's form. These conventions must not be confused with the form of the RAND-ABEL syntax itself. The conventions used here are:

1. The RAND-ABEL language relies on a number of special keywords, or reserved words, which have a particular meaning. Appendix B contains a complete list of RAND-ABEL keywords. As mentioned earlier, in this manual RAND-ABEL keywords are printed in **boldface**, to distinguish them from other language constructs. In a RAND-ABEL program, these keywords must be written in
lower-case, with the first letter optionally upper-case.
Therefore, the only two valid ways of writing a keyword are:

Declare, declare

This same case freedom does not extend to RAND-ABEL identifiers
or character strings. The variable name "Country" is distinct
from the (dangerously similar) name "country".

2. The syntax of RAND-ABEL is sometimes best described in terms of
a set of syntactic categories, which themselves have a defined
structure. These syntactic categories are represented by a
word or phrase in italics, like expression. What is allowed in
place of these categories is described in various sections of
this manual. To find the definition of a category, look under
that category in the index to this manual; it tells on which
page that definition occurs.

3. To represent a set of options, one of which must be chosen, we
use a single-spaced vertical stack of options. For example,

Trace  If.
      Function.

This notation means that the TRACE statement can take the form
"Trace If." or "Trace Function."

4. Ellipses (i.e., "three dots" notation) are used to represent a
sequence of zero or more RAND-ABEL constructs. For example,

statement . . statement

means that zero or more statements can occur in a sequence. By
extension, if a delimiter is used after the first occurrence
and before the second occurrence with the three-dot notation in
between, it means zero or more instances of that construct can
occur in sequence, separated from each other by that delimiter.
For example,
name, . . . , name

means that zero or more RAND-ABEL names can occur, separated by commas. When ONE or more occurrences are required, the above notation is sometimes used for convenience, with a note immediately below the syntax diagram stating that restriction.

5. The structure representing a particular RAND-ABEL language item is boxed, so that it is easily found. Rules specifying various restrictions and notes regarding this structure then follow. The particular syntactic category being (perhaps partially) defined is shown within the top border of the box. For example:

```
statement

If Boolean-expression Then statement

If Boolean-expression Then statement Else statement
```
II. NAMES, IDENTIFIERS, WHITE SPACE, AND COMMENTS

NAMES AND IDENTIFIERS

The entities of a RAND-ABEL program (variables, arrays, etc.) have names. Each is represented by a RAND-ABEL identifier, which is composed of a sequence of one or more of the following characters, without intervening spaces:

- upper and lower case letters (A-Z, a-z)
- digits (0-9)
- the hyphen (-) (or its synonym, the underscore (_))
- the number or pound sign (#)
- the percent sign (%)
- the plus sign (+)
- the period (.)

Rules:

1. An identifier cannot end with a period.
2. An identifier cannot begin with a hyphen (-) or underscore (_).
3. An identifier cannot extend over a line of text (i.e., it cannot contain a carriage-return or line-feed character). It also cannot extend over a line of text by being hyphenated; the hyphen is treated just as any other character. (An identifier used as a column heading in a table statement is an exception to this rule. See the discussion of the table statement in Section VII.)
4. Upper-case letters are distinct from lower-case letters within identifiers; for example, the following identifiers represent different data items:

   Country, country, COUNTRY, CounTry
5. A sequence of characters meeting the above restrictions, and
intended as an identifier, must also not be recognizable as
anything else, such as an integer or real number.

There is no restriction on length, other than the one-line
limitation (Rule 3 above).

It should be noted that various text formatters in use at Rand and
elsewhere interpret a period (.) in column 1 as a special formatting
instruction, thereby causing problems if that is not intended. No
RAND-ABEL statement begins with a period, but one could inadvertently
appear in column 1 if an identifier begins with a period and a statement
is continued onto a following line, causing an identifier to appear
first on the succeeding line. It is safest not to start any identifier
with a period.

Examples of valid RAND-ABEL identifiers:

country
Order-WTVD1-force-assignment
assumptions-re-Europe-On-Call
84flight#-2

An identifier having the form described above can also be an
"identifier constant" as a member of the range of an enumerated data
type. See the section on DATA TYPES for more information on enumerated
data types.

A NOTE ON WHITE SPACE AND COMMENTS

RAND-ABEL statements and definitions consist of a set of words,
some of them reserved keywords, some of them names (of variables,
functions, etc.), and some of them unexecuted noise words or comments,
made up by the program's author. The following rules hold in general in
writing RAND-ABEL programs. Since RAND-ABEL is built upon the C
language, the C conventions for program form should be followed in case
of uncertainty.
Form Rules:

1. One or more spaces separating RAND-ABEL language constructs is considered to be "white space" that acts as a separator. Any comment enclosed by square brackets is also considered to be white space. Examples:

   Let [the variable] Country be US.
   Let [the variable] Country be US.

   Both of these statements are equivalent to the statement:

   Let Country be US.

   White space occurring within a RAND-ABEL table header has special meaning; these rules do not strictly apply within it. See the sub-section "Table Statement" in Section VII for further information.

2. Carriage returns or line feeds are equivalent to space characters in creating white space; they have no other syntactic or semantic meaning. Example:

   Declare [the function] plan by example:
   Let [the] plan of US, [the originator]
   [in] 1984, [the time period]
   [within] Europe [the locale]
   be Defend-borders.

   The above statement is equivalent to the statement:

   Declare plan by example:
   Let plan of US, 1984, Europe be Defend-borders.

   Spaces and other characters occurring within a quoted character string are taken literally, and are not considered white space is the term is being used in this discussion. For further
information about character strings, see Section III, or the syntax of character strings in the C language definition. Also, spaces are treated specially within table headers. See the discussion of the table statement for further information.
III. DATA TYPES

PRIMITIVE DATA TYPES

The RAND-ABEL language recognizes five distinct primitive data types. (In addition, the concept of enumerated data types and the ability to create pointers, functions, and arrays allow the user to construct an arbitrary number of additional data types; see below.) Every simple variable, value, and expression in the language is of one of these data types, either through explicit declaration or (in the case of expressions) by derivation from the form of constants and the prior declaration of variables used within them.

RAND-ABEL is a "strongly typed" language. That is, the data variables, values, or expressions on either side of an assignment statement, or binary operator, or used in place of a function's formal parameter, must agree. This strong typing is possible because all identifiers must be declared explicitly prior to use, thereby associating the identifier with a data type (or in the case of a structure or function, a sequence of data types). The RAND-ABEL translator will flag a statement as being in error if there is a mismatch of data types within the statement. (The only exception to the above statement relates to the various forms of numbers; it is explained below.)

The strong typing goes considerably deeper for constructed data types. Consider the following examples:

- A pointer is declared to point to an array of real numbers. The array has two indices: an integer and an enumerated data type. All assignments of this pointer to other data constructs must retain the characteristics of a two-dimensional array storing real numbers, indexed by an integer and an enumerated data type.
- A function is declared to return a process as its value, and has three formal parameters: a Boolean, a character string, and a pointer to an array having the characteristics given in
the previous example. All calls on this function must meet all these data type constraints, including the correct data types on the indices and stored values of the array pointed to by the third parameter.

The five built-in RAND-ABEL data types are:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Example(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Integer&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Whole number (with no decimal point explicitly represented)</td>
<td>1, -3567, 0</td>
</tr>
<tr>
<td>2. Real&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Decimal numeric value with decimal point explicitly positioned</td>
<td>5.34, -.0079, 0.0, 8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.02 E 23 (= 6.02 x 10^23)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 E 3 (= 4000.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 E -3 (= .004)</td>
</tr>
<tr>
<td>3. String&lt;sup&gt;c&lt;/sup&gt;</td>
<td>String of zero or more characters delimited by quotation marks.</td>
<td>&quot;Ally is not responsive.&quot;</td>
</tr>
<tr>
<td>4. Boolean</td>
<td>A logical data item which can only take on one of the two values: Yes, No</td>
<td>Yes, No</td>
</tr>
<tr>
<td>5. Process&lt;sup&gt;d&lt;/sup&gt;</td>
<td>An identifying number for an RAND-ABEL co-process. Two reserved RAND-ABEL keywords represent the current process (Self) and its parent (Parent).</td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup>Integers are implemented in RAND-ABEL as the C language data type "long int", and are therefore subject to C restrictions for that data type.

<sup>b</sup>These numbers must have a decimal point contained in them, or use "E" (exponential) notation to represent a power of 10. If they use "E" notation, the number following the "E" must be a whole number. Since use of the E operator could be construed as part of an identifier (e.g., in 4.0E3 or even 4.0E3), it must be separated from its arguments by white space. Real numbers are implemented in RAND-ABEL as the C language data type "real", and are therefore subject to C restrictions for that data type.

<sup>c</sup>A string may be up to 256 characters in length, and may contain embedded carriage returns, line feeds, and so forth. The normal syntax and rules for character strings in the C language apply for character strings in RAND-ABEL.

<sup>d</sup>Co-processes are described in Section X.
ENumerated Data Types

In addition to these primitive data types, additional data types may be constructed by the following mechanisms: enumerated data types, arrays, and pointers to various data constructs.

Enumerated data types may be constructed by declarations of the following type:

Declare country by example: Let country be {France, Germany}.

This defines a new data type, country, that can take on exactly two values: the identifiers France and Germany. These values will be called "identifier constants" within this document, and the data type (in this case, country) is sometimes called an enumerated variable.

Enumerated data types take on an explicit, finite-ordered list of values. The values are simple RAND-ABEL identifiers.

The only way an identifier constant is established is by the declaration of an enumerated variable including that identifier constant in its range (i.e., within the braces {}). The set of valid identifier constants is given in a declaration of the form:

Declare color by example: Let color be {Red, Green, Blue}.

In this example, "color" is a new data type, and "Red", "Green" and "Blue" are the (only) identifier constants in its range. (See Section V for a more complete description of declaration options.)

There is one reserved enumerated value that any enumerated data type can take on: Unspecified. This is not explicitly declared as a value. Note that it is NOT automatically assigned by RAND-ABEL to any variable, e.g., prior to an explicit assignment of a value to an enumerated-type variable. This value can be explicitly given to an enumerated-type variable through use of the keyword Unspecified (or its synonym "--") used in place of an identifier constant (e.g., in an assignment statement).
All enumerated data types are distinct from one another. Therefore, the value of one enumerated data type cannot be assigned to another (because this violates the "strict type checking" rule that only the same data types may be compared, assigned, or operated on together). Furthermore, all identifier constants are distinct from each other. For example, consider the two declarations:

Declare color by example:
Let color be (Red, Green, Blue).

Declare mood by example:
Let mood be (Happy, Angry, Blue, Querulous)

If the two assignment statements were executed:

Let color be Blue.
Let mood be Blue.

then not only is it NOT true that the value of color equals the value of mood (because the two Blues are distinct identifier constants), BUT THEY CAN'T EVEN BE COMPARED, as in:

If color = mood Then ...

because, as stated earlier, color and mood are two distinct data types, and therefore it is illegal to compare them.
IV. VALUES, EXPRESSIONS, AND OPERATORS

VALUES AND SIMPLE EXPRESSIONS

In a programming language, a value is informally considered to be a simple term that can be evaluated to yield either a storage location or the contents of that location. If it appears on the left-hand side of an assignment statement, its evaluation yields a location at which the assignment is to be made; if it appears on the right-hand side of an assignment, or elsewhere, its evaluation yields a data value. The description of the syntax of RAND-ABEL relies on the following categories of values and expressions, which are explained in this section:

<table>
<thead>
<tr>
<th>Name</th>
<th>Informal Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>array-access</td>
<td>Access to the value stored at one of the cells in an array.</td>
</tr>
<tr>
<td>lvalue</td>
<td>A reference which can appear on the left-hand side of an assignment statement; that is, it designates a storage location at which a value is located.</td>
</tr>
<tr>
<td>unitparam</td>
<td>A value that is assigned to a parameter in a function call. A simple value, or else a parenthesized expression.</td>
</tr>
<tr>
<td>simple-expr</td>
<td>Any of the above values, or in addition a pointer to a function.</td>
</tr>
<tr>
<td>expression</td>
<td>A value, or a sequence of values related by operators.</td>
</tr>
</tbody>
</table>

The following tables give more precise definitions for these terms, nested to show the manner in which some definitions include others. For example, since the boxes defining unitparam and lvalue are contained within the box labelled simple-expr, all of the varieties of unitparam
and lvalue (defined by the contents of their boxes) can be used wherever a simple-expr is needed. Similarly, the various types of array-access can be used as whenever an lvalue is needed.

expression

**Report from** function-invocation

**Evaluate** unitparam . . . unitparam
   with format-spec unitparam . . . unitparam

numeral variable-name
unary-operator expression
expression binary-operator expression

**There is** array-access
   a array-access
   an array-access

**Describe** array-name

simple-expr

Rules:

1. The **Report from** function-invocation expression calls the named function and returns as its value the value returned by the function. Example:

   Let message be Report from plan of US, 1984, Europe.

2. The **Evaluate** expression allows the RAND-ABEL writer to generate a string of characters (usable in further RAND-ABEL processing) from a sequence of arguments, each of them a unitparam. This string may then be used in subsequent RAND-ABEL statements--for example, as an argument to a function that requires a string as one of its parameters. The format-spec is a string of characters that can control the formatting of the resultant string; the syntax and options available for a format-spec are described in the subsection "Input/Output" in Section VII. Example:
Perform data-logging
using Evaluate "The ally of" country "is" (ally of country)
as message.

In the above example, blanks are not required within the
code strings to prevent the value of country from running
into "The ally of" and "is" because the default print format
for an enumerated variable contains prefix and postfix blanks.
(See "Default Output Formats" in Section VII.)

3. The expression of the form numeral variable-name is a
restricted form of implied multiplication. For example, one
might use the phrase

3 kilometers

in a place where distance in meters is required; assuming the
variable named "kilometers" had the value 1000, one would
obtain the desired result. (This is a dangerous construction
and should only be used under highly controlled circumstances.)

4. There is ... evaluates to Yes or No depending on whether the
named array element has a stored value within an erasable
array. (If the array has not been mentioned in a Make Erasable
or Make Semi-erasable statement, it is an error.) Example:

If There is a border of Iran, Syria
Then Perform relative-strength-test using
  Iran as country_1 and
  Syria as country_2.

5. Describe array-name returns a character string that describes
all the combinations of indices and their corresponding values
for which the array is presently defined. It is used primarily
for list-stored (i.e., Erasable) arrays (see "To Declare an
Array...") and for debugging. However, it will work for all
variables and arrays, regardless of method of storage. A
variable is equivalent to an array with zero indices (a "0-ary
array”). The separate linguistic construct called "variable" is only used for convenience in referring to this common construction.

A **simple-expr** is defined by the following nested set of tables. The nesting again shows that certain terms are contained within the definitions of others. For example, a **unitparam** is one valid type of **simple-expr**, so any of the methods of constructing a **unitparam** can be used wherever a **simple-expr** is needed.
**simple-expr**

**Function** *function-ptr*

*function-name*

**unitparam**

Yes

No

@c-code@

enumerated-value

numeric-literal

quoted-string

*variable-name*

Unspecified

--

( *expression* )

**lvalue**

*variable-name*

Occupyant of *variable-name*

array-access

Address of *variable-name*

Function *function-name*

Attribute *array-name*

**array-access**

*simple-expr* array-name *simple-expr*

array-name of *simple-expr*, . . .

in , and

by , and

. . . , *simple-expr*

and , and
Rules:

1. The **Function** keyword, followed by the name of, or a pointer to, a RAND-ABEL function returns as its value the address of that named function, or the address of the function pointer referred to by the **function-ptr**. This returned value is of type pointer.

2. The "@" sign provides an escape to C code; see the C statement description in Section VIII for more information.

3. "--" is a synonym for the keyword **Unspecified**, which is used in conjunction with enumerated data types. (See the discussion of enumerated data types in Section III.)

4. The variable-name or array-access following the keywords **Occupant of** must be of data type pointer. The clause returns the contents of the storage location pointed at by that pointer.

5. The **Address of** clause yields a value of type pointer. For example:

   \[ \text{Address of Attribute country-array} \]

   returns a pointer to the array named "country-array".

6. The first type of array access:

   \[ \text{simple-expr array-name simple-expr} \]

   is used only for arrays with two indices. To be accessed in this infix manner, with the array name appearing between the indices, the array must have been declared in that same manner, as in:

   \[ \text{Declare larger-than by example:} \]
   \[ \text{Let Iran larger-than Iraq be Yes.} \]
Arrays declared in an infix manner must always be accessed in an infix manner. The second, "prefix", form of array access can be used with any number of indices (including two), and similarly must be declared using the prefix form:

Declare between by example:
Let between of Canada, US, Mexico be Yes.

Arrays declared in a prefix manner must always be accessed in a prefix manner.

OPERATORS

The operators used to construct expressions can be categorized as numeric operators, comparison operators (providing equality and inequality tests), logical operators (yielding a Yes or No result), and string operators. Each of these categories is described below. In each case, we list a "preferred form" for representing these operators, to create as much consistency and readability in RAND-ABEL programs as possible.

Numeric Operators

<table>
<thead>
<tr>
<th>Mathematical Notation (Preferred Form)</th>
<th>&quot;English-like&quot; Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>plus</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>times</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>divided by</td>
<td>Division</td>
</tr>
<tr>
<td></td>
<td>modulo</td>
<td>Modulo</td>
</tr>
<tr>
<td>-</td>
<td>negative</td>
<td>unary &quot;-&quot; sign</td>
</tr>
</tbody>
</table>

When an integer variable accepts the result of the division of two integers, the result will be truncated toward zero to an integer.

Examples:
73/10 evaluates to the value 7
-73/10 evaluates to the value -7

Division by zero results in a run-time error.
These operators can yield floating point exceptions (i.e., error conditions) in a machine-dependent manner.
The modulo operator returns the remainder upon division. Example:

23 \textit{modulo} 8 \textit{is} 7.

The modulo operator requires integer arguments.
Only integer and real data types may be used as arguments for these operators. (Exception: the more stringent requirement for modulo arguments mentioned above.)

\textbf{Comparison Operators}

Comparison operators are categorized below as "equality" or "inequality" type operators. All comparison operators yield a value that is Boolean. One of the operands to these comparison operators can have an ambiguous data type, IF that ambiguity can be resolved by the requirement of consistency with the other operand's data type.
RAND-ABEL does not, however, accept two operands of ambiguous data type, and attempt to resolve the mutual ambiguity. (Ambiguity can arise from a value such as the enumerated constant Blue, if more than one enumerated data type contains this constant in its range.)

\textbf{Equality Tests}

<table>
<thead>
<tr>
<th>Mathematical Notation (Preferred Form)</th>
<th>&quot;English-like&quot; Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{=}</td>
<td>is</td>
<td>Is equal to</td>
</tr>
<tr>
<td>\texttt{\sim=}</td>
<td>is not</td>
<td>Is not equal to</td>
</tr>
</tbody>
</table>

\texttt{are not}
Both arguments to these operators must have the same type (with one exception: an integer appearing where a real number is needed is interpreted as a real number for that purpose).

Two operands from any one data type may be compared using these equality operators.

Two strings are equal only if they have the same length (including possibly zero length--i.e., the null string), and at each respective character position, their corresponding characters are equal. (Upper case and lower case versions of a character are treated as different characters in this test.)

Two enumerated values are equal only if they are represented by the same identifier constant, and are in the range of the same (enumerated) data type. The reserved word Unspecified is equal to Unspecified for the same data type.

It is an error if two enumerated values are compared that belong to different (enumerated) data types.

### Inequality Tests

<table>
<thead>
<tr>
<th>Mathematical Notation (Preferred Form)</th>
<th>&quot;English-like&quot; Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=</td>
<td>is at least</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is at most</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>is less than</td>
<td>Less than</td>
</tr>
</tbody>
</table>

Both arguments to these operators must have the same type (with one exception: an integer appearing where a real number is needed is interpreted as a real number for that purpose).

The following data types may be compared using the inequality operators: integer, real, string, enumerated. Note that Boolean data types CANNOT be compared using these operators.
Integer and real data types compare according to their values. Strings a and b compare as follows. (In comparing two individual characters, the collating sequence is used for the individual computer on which RAND-ABEL resides; this test is therefore implementation dependent.)

a. If a and b are both the null string, they are equal.
b. If one of them is the null string and the other is not, then the null string is less than the other.
c. Otherwise, compare strings a and b character-by-character; if each of these comparisons is Yes (i.e., true) at the time the end of one of the strings is reached, but the other string still has additional characters, then the shorter string is less than the longer one.
d. Otherwise, compare strings a and b character-by-character; if each of these comparisons is Yes (i.e., true) up to character position k, but is No (i.e., false) at character position k + 1, then if string a's character in position k + 1 is less than, or greater than, string b's character in position k + 1, then string a is less than, or greater than (respectively) string b.

Enumerated values compare with the inequality operators according to the following rules:

a. If either or both values is Unspecified (or its synonym "-"), then the result is No.
b. If the values belong to different (enumerated) data types, it is an error.
c. Identifier constant C1 is less than identifier constant C2 if and only if C1 appears before C2 in the sequence of identifiers defining the range of their common (enumerated) data type. If they are the same identifier within this data type, they are equal.
Logical Operators

Logical operators are used to combine two different Boolean operands—that is, ones taking the values Yes or No—to yield a new Boolean value. For example, a RAND-ABEL program might require the logic:

If agreement and (Red-violates or Blue-violates)
Then Let agreement be No.

The assignment of the value No to the variable "agreement" will take place only if the existing value of agreement is Yes, and in addition either "Red-violates" or "Blue-violates" (or both) is Yes.

<table>
<thead>
<tr>
<th>Mathematical Notation</th>
<th>&quot;English-like&quot; Notation (Preferred Form)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>and</td>
<td>Logical &quot;and&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>~</td>
<td>not</td>
<td>(unary) Logical &quot;not&quot;</td>
</tr>
</tbody>
</table>

The meanings of these operators are given in the following table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>not a</th>
<th>a and b</th>
<th>a or b</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

The logical operators require Boolean values (that is, Yes or No) as their arguments, and return a Boolean value as the result.
String Operator

The single string operator concatenates two strings to yield one resulting string. Concatenation may be used, for example, in the creation of tailored messages, as in:

Let outstr be "WARNING: "$ message "$ PLEASE RESPOND (Y/N): ".

In this example, the string variable "outstr" receives a string containing a variable "message", along with standard prefix and suffix strings.

<table>
<thead>
<tr>
<th>Mathematical Notation (Preferred Form)</th>
<th>&quot;English-like&quot; Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ concatenated with</td>
<td></td>
<td>Concatenation</td>
</tr>
</tbody>
</table>

Only string values may be concatenated together. The result is a string consisting of the first string followed by the second string.

Example: "This is " $ "a test."

and

"This is "$ "a test."

and

"This is concat"enated with "a test."

are all equivalent to:

"This is a test."

Whenever there is any ambiguity or uncertainty, parentheses should be used to specify the order in which operators should be applied within an expression. When more than one operator is used in a sequence, precedence relations are used to resolve the order. Operators with higher precedence are performed first; within the same precedence, operators are performed within the expression from left to right. Operators of the same precedence associate to the left. For example,
(a & b & c) = ((a & b) & c).

The following table gives the precedence of RAND-ABEL operators. Operators in the same row are of equal precedence.

Highest precedence: ~ (not)  - (unary)  modulo
                     *  /  modulo
                     +  -
                     <  >  <=  >=
                     =  ~=
                     & (and)

Lowest precedence: | (or)
V. DECLARATIONS

TO DECLARE A VARIABLE

\[
\text{Declare variable-name by example:}
\]
\[
\text{Let variable-name be expression.}
\]
\[
\{ \text{name . . . name } \}.
\]
\[
\{ \text{name , . . . , name } \}.
\]

Rules:

1. The type of the variable becomes the same as the type of the expression.
2. The type of the expression must be uniquely determinable at the time the statement is encountered. (For example, if the same enumerated value appears in the range of several enumerated data types, then it may not be used in an assignment within a declaration.)
3. The form \((\text{name . . . name})\) (with or without the optional commas as separators) creates an enumerated data type, and declares the specific values that variable can acquire (in addition to the special enumerated value Unspecified), with each value an identifier constant. Examples:

Declare troop-strength by example:
\[
\text{Let troop-strength be 10000.}
\]

Declare force-ratio by example:
\[
\text{Let force-ratio be 5.8.}
\]

Declare message by example:
\[
\text{Let message be "Help!".}
\]

Declare agreement by example:
\[
\text{Let agreement be Yes.}
\]
Declare current-force-test by example:
Let current-force-test be Address of Function calc1.

Declare alliance by example:
Let alliance be (France, Germany, Spain).

The spatial alignment of these statements is not important; they are aligned here by variable name merely for ease in reading.

TO DECLARE AN ARRAY; MAKE AND ERASE STATEMENTS FOR ARRAYS

Array Declarations

Arrays are declared and referenced in two possible ways: infix and prefix. "Infix" refers to placing the name of the array between its two indices; "prefix" means placing the array name before its list of indices.

If an array has exactly two indices, it may be declared in an infix manner; if it is, it must then always be referenced in the same infix manner. All arrays may be declared (and then necessarily referenced) in a prefix manner.

Note that a one-dimensional array, indexed by the smallest positive integers (1, 2, 3, ...), may be called a "vector" in other computer languages. A two-dimensional RAND-ABEL array indexed by the smallest positive integers corresponds with the term "matrix" in other computer languages.

The syntax diagrams below show the two different ways of declaring a RAND-ABEL array.
Prefix form:

```
declaration

Declare array-name by example:

Let array-name of simple-expr, . . .
in and
by . . .
and
... , simple-expr
and
, and

be expression.

{ name . . . name }.
{ name , . . . , name }.
```

Infix form:

```
declaration

Declare array-name by example:

Let simple-expr array-name simple-expr

be expression.

{ name . . . name }.
{ name , . . . , name }.
```

Rules:

1. In the first (prefix) form there must be at least one simple-expr.
2. The second (infix) form is used only for arrays with two indices, as an option. Such an "infix array" is a separate data type, and must always be accessed as an infix array.
3. The type of the array is the type of the expression, which must be determinable at the time the statement is encountered.
4. Arrays can have one or more indices. (Arrays with zero indices are equivalent to variables.)

5. The simple-exprs that are used to index the array must be either of type integer or enumerated.

6. If any index is of type integer, it is designated by a single integer constant, \( n \), in place of simple-expr. This index can then take on the integral values \( 0 \ldots n \). Note that index \( n \) means that the index can take on \( n + 1 \) distinct values.

7. The form \( \{ \text{name} \ldots \text{name} \} \) (with or without the optional commas as separators) creates an array of enumerated data type, and declares the specific values that variable can acquire (in addition to the special enumerated value Unspecified), each value being an identifier constant. Examples:

Prefix array with integer indices and enumerated value:

\[
\text{Declare chessboard-square by example:} \\
\text{Let chessboard-square of 7 and 7 be} \\
\{\text{king, queen, knight, bishop, rook, pawn, empty}\}.
\]

Infix array with enumerated indices and Boolean value:

\[
\text{Declare Borders-on by example:} \\
\text{Let US Borders-on Canada be Yes.}
\]

Infix array with enumerated indices and integer value:

\[
\text{Declare Length-of-border-with by example:} \\
\text{Let US Length-of-border-with Canada be 5000 [km].}
\]

Erasable Arrays

There are times when selected values of an indexed array are to be stored and retrieved, but it is wasteful or impossible to allocate storage for all possible array values. For example, if the array "posture" takes four indices—a country (one of 50 enumerated values), a "readiness index" (one of 20 integers), an opponent country (one of 50 enumerated values), and a morale factor (one of 10 integers)—then array storage must be set aside for \( 50 \times 20 \times 50 \times 10 \) values, or 500,000 locations. Only a small fraction of these might be used in practice. Arrays having only a small fraction of their values set are commonly referred to as "sparse arrays".
Sparse arrays may be stored in list structures, rather than arrays. List structures require storage space proportional to the number of cells used, not the number of potential cells. However, access to array cells in a list-stored array is slower than direct indexed access to an explicitly stored array.

In RAND-ABEL, sparse (i.e., list structured) storage of an array is requested through the `Make` statement:

```plaintext
Make array-name Erasable.
    Unerasable.
    Semi-erasable.
```

Rules:

1. The `Make` statement must immediately follow the `Declare` statement for the array.
2. The `Erasable` option instructs RAND-ABEL to store the named array in a list structure, as a sparse array. Attempted access of an `Erasable` array element before it is set is a run-time detectable error.
3. The `Unerasable` option instructs RAND-ABEL to store the named array as an explicit indexed array (i.e., as arrays are stored in the C language). Access of an `Unerasable` array element before it is set is not detected, and can result in indeterminate (and therefore highly dangerous) results.
4. The `Semi-erasable` option is used for debugging. The array will be list stored, and the "There is" expression will be available for it (see below), but the `Erase` statement (described below) will not be available for it. For a `Semi-erasable` array, access of an array element before it is set is a run-time detectable error; this option is therefore useful in detecting errors in programs during their development.
If no **Make** statement is given for an array, RAND-ABEL determines the storage mode for the array based on its size and available storage space. This decision could vary between runs of the RAND-ABEL Translator.

The storage method RAND-ABEL uses for an array has only three effects detectable by the user:

1. The speed of array accesses can vary; for programs heavily dependent on arrays, this could affect program efficiency.
2. For list-stored arrays, attempted access of an array element before it is set is a run-time detectable error; for traditionally stored arrays, the error is not detected and an indeterminate value is accessed.
3. Array elements in a list-stored array may be "erased" and the storage thereby reclaimed. This option is not available for traditionally stored arrays. (Since this is a main operational difference between list-stored and traditionally-stored arrays, the keywords **Erasable** and **Unerasable** are used in the **Make** statement to distinguish between list storage and traditional storage, respectively.)

Erasure of all elements, or a selected element, of a list-stored array is accomplished with the **Erase** statement:

```plaintext
Erase array-name.
array-access.
```
Rules:

1. Only arrays that have been made *Erasable* can be used in conjunction with the *Erase* statement.
2. If an *array-name* is given, all currently stored elements of that array are erased, and the storage reclaimed.
3. If an *array-access* is given, accessing an individual element of the array, then only that element is erased, and the storage reclaimed.

A Boolean expression is available for testing the existence of array elements in *Erasable* arrays:

\[
\text{There is a } \text{array-access}.
\]

See the section "Values, Operators, and Expressions" for a description of the operation of the *There is* expression.

**TO DECLARE A FUNCTION**

Every function that is used must be declared. Every function either always returns a value, or never returns a value. The function declaration indicates which of these cases applies, as well as the data type of the arguments and value returned, if any.

```
declaration

Declare func-name by example:

Let expression be Report from named-function-call.
Perform named-function-call.
```
Rules:

1. The first form must be used when the function returns a value. The type of the expression must be the same as the type of value returned by the function. The type of the expression must be determinable at the time this statement is executed.

2. The second form is used only when a function does not return a value.

3. A function must be declared before it is defined, and it must be defined before any use. Section VI discusses function definitions.

A named-function-call is one that explicitly uses the function name to invoke it, not a pointer to that function. Section VI describes the named-function-call.

Examples:

Declare select-country by example:
   Let France be Report from
       select-country using alliance as range
           and strength as criterion.

Declare force-calc by example:
   Let 5.0 be Report from force-calc using
       France as country.

Declare validity-check by example:
   Perform validity-check.
VI. FUNCTIONS

DEFINING A FUNCTION

```
function-definition

Define func-name : declaration . .
    declaration
    statement . .
    statement
End.
```

A declaration is any of the declaration types (starting with the keyword Declare) listed in Section V.

If the function returns a value, at least one of the statements within the function definition must be "Exit Reporting simple-expr". Moreover, one such statement must be reached during execution of the function, otherwise a run-time error will occur.

If the function does not return a value, it is exited either by an explicit Exit statement, or else by "falling through" the statements to the End statement.

Examples:

```
Define Timed-wakeup:
    If Time is at least Time-to-wake of (Command-ID of self)
    Then
        ( Record "Starting move at maximum time = " Time ".".
        Exit Reporting Yes.
    )
    Else Exit Reporting No.
End.
```
Define Next-move:

Declare piece by example: Let piece be bishop.
Let piece be whites-last-move-piece.
Let y be col-location of piece.
Let x be row-location of piece.
If piece = pawn
Then
{ If (Occupant of y and x+1) = empty and
   (Occupant of y+1 and x+1) = empty and
   (Occupant of y-1 and x+1) = empty
   Then
   { Let row-location of piece be x+1.
     Exit Reporting "Done."
   }
   [ Place other piece moves here as they are programmed. ]
 }
Else Exit Reporting "Can't handle."
End.

Note that in the above example, the scope of certain variables such as x and y is global (i.e., externally declared in the Data Dictionary outside of the function definition).

NAMED FUNCTION CALLS AND FUNCTION INVOCATIONS

A named-function-call is an invocation of a function in which the name of the function appears explicitly. It is required, for example, as part of the declaration of that function (which announced the names and data types of its arguments, and the type of its returned value, if any).

```
func-name

func-name using expression as param-name, ...
  for and
  , and
  ..., expression as param-name
  and for
  , and
```
Rules:

1. In a named-function-call, the func-name must be given explicitly; a pointer to a function is not allowed in this case.
2. When used as an example in a function declaration, the types of each expression must be determinable at the time the declaration is encountered.

By contrast, a function-invocation has the same form as a named-function-call, but it can have a pointer to a function in place of an explicit function name:

```
function-invocation
  named-function-call
  func-ptr
  func-ptr using expression as param-name, ...
    for and
    and
    ... expression as param-name
    and for
    and
```

When a function does not return a value, it is invoked through the statement

```
Perform function-invocation.
```

When a function returns a value, it is accessed via the expression

```
Report from function-invocation.
```
Examples:

Let message be Report from next-move
    using pawn as whites-last-move-piece.

If Report from Timed-wakeup
    using now as time is Yes
Then Perform Work.
Else Perform Error-handler.
VII. RAND-ABEL STATEMENTS

Statements are used in RAND-ABEL to define the operation of a function. (Several statements can also occur at the "top level" in RAND-ABEL outside of a function definition to set the global context in which other statements will operate: They are declarations, the Data Dictionary (see Section IX), the C statement, and the Trace and Untrace statements.)

The various forms of RAND-ABEL statements are described below within the following categories:

- Assignment
- Conditional execution
- Repetitive execution
- Functions: invoking and exiting
- Input/output
- Compound and null statements
- Table statement

All RAND-ABEL statements begin with a keyword which uniquely identifies the statement type. In general, all RAND-ABEL statements end with a period; the only exceptions are compound, conditional, and repetitive statements whose form has an embedded statement as the last entity within the form. In those cases, the period ending the embedded statement becomes the statement delimiter.

ASSIGNMENT

Assignment statements are used to store a value in the location specified by either a variable or an array element.
Let \( lvalue \) be \( expression \).

pointer

**Increase** \( lvalue \) by \( expression \).

Decrease

Multiply

Divide

Rule: The (data) types of the terms on the left-hand side and right-hand side of the assignment statement must match.

Two exceptions:

1. If a real number is required by the left-hand side, then if the value of the \( expression \) is integer, that integer is coerced into a real number for the purpose of this statement.

2. The right-hand side can be an enumerated identifier constant of ambiguous data type if that ambiguity is resolved by the type of the \( lvalue \) or \( pointer \) on the left-hand side.

The terms \( lvalue \) and \( expression \) are defined in Section IV. In general, an \( lvalue \) is what can normally occur on the left-hand side of an assignment statement: namely, a term giving the address of a named storage location, not a pure value.

Examples:

Let gross-profit be gross-sales - cost-of-sales.

Let force-ratio be Report from force-calc
using France as side-1 and Yugoslavia as side-2.

Decrease force-ratio by 2.5.
CONDITIONAL EXECUTION

Conditional execution is controlled by the If statement. It allows certain RAND-ABEL statements to be executed only if certain conditions are true, or are false.

```
statement
  If Boolean-expression Then statement
  If Boolean-expression Then statement Else statement
```

Rules:

1. The Boolean-expression is any expression that evaluates to type Boolean--i.e., that takes on values Yes and No.
2. If the Boolean-expression evaluates to Yes, then the first statement is executed.
3. If the Boolean-expression evaluates to No and the Else clause is present, the statement following the Else keyword is executed. If the Boolean-expression is No and no Else clause is present, no action is taken.
4. As is normal programming language practice, if conditional statements are nested, an Else clause is attached to the nearest previous If clause that does not yet have an Else clause attached. If one needs a null If or Else clause to keep the logic straight, use the RAND-ABEL null statement, consisting of just a period, as in:

   If king-uncHECKED Then. Else Perform Think.
   or
   If king-in-check Then Perform Think. Else.
This statement is not delimited by a period, for reasons given at the beginning of this section: the last statement embedded within the If statement will contain its own delimiter.

Either statement can of course be a compound statement (that is, one or more declarations and statements contained within "{" and "}") braces) thereby allowing any needed complexity in logic to be stated in the Then or Else clauses.

Example:

If user-response = "Y" or user-response = "YES"
Then
  { Perform Recalculation.
    Print "Calculation Completed. More? (Y/N): "
    Let user-response be Report from query-user.
  }
Else If user-response = "N" or user-response = "NO"
Then
  { Print "No action taken. More? (Y/N): "
    Let user-response be Report from query-user.
  }
Else If user-response = "?"
  Then Perform Help-function.
Else
  { Print "Your response not understood."
    Perform Help-function.
}

Note that If ... Then rules can also be formed using the table statement.

REPETITIVE EXECUTION

The RAND-ABEL For and While statements allow one or more statements to be executed repetitively—that is, zero or more times, depending on the controlling variable or expression.
statement

For variable : statement
While Boolean-expression : statement

Rules:

1. In the first form, the variable must be of the enumerated data type. The statement is executed once for each identifier constant in the range of the variable, with the variable bound in turn to each identifier constant, in the order in which the identifier constants are declared as being the range of the enumerated data type. Examples:

Declare alliance-members by example:
Let alliance-members be
     (France, Germany, Spain).

For alliance-members Perform Force-calc.

For each-country (US or UK or FRG or Belgium):
{
    Let Membership of each-country be Nato.
    Let Side of each-country be Blue.
}

2. In the While form, the Boolean-expression is evaluated. If its value is Yes, the statement is executed; if the value is No, no further action is taken. If the statement executes, the Boolean-expression is then reevaluated, and if Yes the statement is reexecuted. This sequence continues until the value of the Boolean-expression becomes No. Example:
Let $k$ be 3.  
While $k > 0$: {
    Print resultsfile $k$.  
    Decrease $k$ by 1.  
}

...leads to the following records sent to the resultsfile:

3
2
1

The following two RAND-ABEL statements are used within a repetitive execution to change the flow of the program's logic:

- `Continue`.  
- `Break`.  

Rules:

1. Within a repetitive execution, the `Continue` statement acts as completion of the current repetition, and control passes to the next repetition of the loop, if any.
2. The `Break` statement acts as completion of all repetitions of the loop, and control passes to the statement following the repetitive statement.
3. In both cases, control returns to the most immediately inclusive `Table`, `For` or `While` statement. That is, to the innermost repetitive statement if they are nested. Examples:
Let \( k \) be 3.

While \( k > 0 \):

\[
\begin{align*}
&\text{If } k = 1 \text{ Then Break.} \\
&\text{Else (Print resultsfile } k. \text{ Decrease } k \text{ by 1.)}
\end{align*}
\]

...leads to the sequence of records in resultsfile:

3
2

Let \( k \) be 3.

While \( k > 0 \):

\[
\begin{align*}
&\text{If } k = 2 \text{ Then Continue.} \\
&\text{Else (Print resultsfile } k. \text{ Decrease } k \text{ by 1.)}
\end{align*}
\]

...leads to one printed record in resultsfile:

3

...followed by an infinite loop, with \( k = 2 \) and the While statement repetitively executing with no effects.

Repetitive execution can also be achieved by the RAND-ABEL table statement. This special RAND-ABEL statement is described in the following section.

**TABLE STATEMENT**

The **Table** statement is the most powerful statement in RAND-ABEL. It can be used to call a function repeatedly, with different arguments, or as a decision table. It is an example of a statement with a two-dimensional syntax; the spatial layout of the table-header is important in determining the meaning of the table statement.
Basically, the table statement allows the named function or the compound-statement to be executed once for each row of data in the table-body. If the table statement contains a named function, the columns of data within the table-body are matched up with the function's formal parameters by means of the column headings within the table-header; if the table statement contains a compound-statement, the local variables declared within the highest level block of that compound statement are matched with the columns of data within the table-body by means of the column headings within the table-header.

The concept and power of the table statement is best illustrated by example. The following RAND-ABEL table uses the function "Deploy". It is an expanded version of the example shown in Section I of this Note.
### Table Deploy

[This table initiates the deployment of assigned forces to the Central European theater]

<table>
<thead>
<tr>
<th>qty</th>
<th>#-%</th>
<th>unit-type</th>
<th>unit-owner</th>
<th>assigned-to</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>in-area</td>
<td>to-area</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>Denmark</td>
<td>CEur</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>CEur-1</td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>Netherlands</td>
<td>CEur</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>CEur-2</td>
</tr>
<tr>
<td>25</td>
<td>%</td>
<td>Troops</td>
<td>FRG</td>
<td>CEur</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>CEur-3</td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>UK</td>
<td>CEur</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>CEur-4</td>
</tr>
<tr>
<td>100</td>
<td>%</td>
<td>Troops</td>
<td>Belgium</td>
<td>CEur</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>CEur-5</td>
</tr>
</tbody>
</table>

This table statement causes the function Deploy to be called five times, once for each row of the table-body. (Each row has seven entries, the last two being "folded over" so that they appear underneath the columns labelled "unit-owner" and "assigned-to").

Another important use of a table statement is as a decision table. It is standard practice (although not required by the syntax of the table statement) to use a slash (/) within the table header to separate the conditions from the action to be taken. (Formerly, a vertical bar (|) was commonly used, but this symbol has taken on a new meaning within a table header, and can no longer be used as a "white space"; see below.) For example, consider the following decision table used as the "Then" clause of a conditional statement:
If Current-situation is Eur-demo-tac-nuc

["Eur-demo-tac-nuc" represents the situation in which one or both superpowers have used some tactical nuclear weapons in Europe, but have done so primarily for demonstrative purposes, i.e., to coerce the opponent into terminating]

Then
{
Table
{
Declare Basic-status# by example: Let Basic-status# be Basic-status.
Declare Risks# by example: Let Risks# be Risks.
Declare Escalation-guidance# by example: Let Escalation-guidance# be Escalation-guidance.

If (Basic-status# is Basic-status or
    Basic-status# is Unspecified) and
(Risks# is Risks or Risks# is Unspecified)
Then
{
    Let Escalation-guidance be Escalation-guidance#.
    Break.
}
}

<table>
<thead>
<tr>
<th>Basic-status#</th>
<th>Risks#</th>
<th>Escalation-guidance#</th>
</tr>
</thead>
<tbody>
<tr>
<td>goals-met</td>
<td>--</td>
<td>Eur-term</td>
</tr>
<tr>
<td>progress-good</td>
<td>low</td>
<td>Eur-demo-tac-nuc</td>
</tr>
<tr>
<td>progress-marginal</td>
<td>low</td>
<td>Eur-gen-tac-nuc</td>
</tr>
<tr>
<td>progress-good</td>
<td>marginal</td>
<td>Eur-demo-tac-nuc</td>
</tr>
<tr>
<td>progress-marginal</td>
<td>marginal</td>
<td>Eur-gen-tac-nuc</td>
</tr>
</tbody>
</table>

Note the use of the Break. statement within the compound statement defining the operation of the Table statement, in order to stop the iteration through the table rows as soon as a satisfactory condition is found.

Much more succinct and powerful means of representing decision tables are being introduced into the next version of RAND-ABEL. Shapiro et al. (1985) contains a brief discussion of these ideas. Using these new table constructions, the table above reduces to the succinct:
As a special case of the new table features, but one that greatly improves the readability and writability of RAND-A2EL programs, the above decision table has been made available in the currently operational version of the language.

The rules for constructing a table-header are as follows:

1. A table-header consists of one or more "text islands", each representing the name of a parameter (if a function is named) or the name of a local variable (if a compound-statement is used).
2. A "text island" is a two-dimensional grouping of characters such that each character of the group is directly adjacent (either horizontally or vertically, not diagonally) to some other character in the group.
3. In addition to hyphens (-) that occur as part of the identifier representing a parameter or variable name, dashes can be used as hyphens to continue the text within a text island onto the next line. These additional hyphens serve only as continuation characters, and are not considered part of the identifier itself. Note in the above example that the column heading:

\[
\text{amo-} \quad \text{unt}
\]

would match with the formal parameter "amount" in the definition of the function "Deploy". Note also that spaces are not permitted within an identifier used as a column header, so
indications of ownership in such an identifier (e.g., "Red's Presumed-opponent") are not permitted.

4. In addition to normal "white space" characters (space, tab, new line) and comments (enclosed in square brackets), the following characters are also considered "white space" in determining the "text islands" comprising a table header:

```
\ / _ =
```

5. If a "connector character" is useful in retaining the integrity of a "text island", the following characters may be used. They provide the adjacency required by Rule 2, but are not themselves considered part of the identifier represented by the text island:

```
| ()
```

(These connector characters also "count" in determining the ordering of the text islands; that is, their position as part of a column heading helps determine the relative position of that column heading.)

6. At times it is desirable to abbreviate a formal parameter's name, or split it between two or more lines. An apostrophe (') may be used as a "wild card" character to match zero or more characters in the parameter name. Also, additional hyphens may be used to make a name split between lines more readable, as described in Rule 3. For example, a formal parameter entitled "Responsible-Party-Action" might be represented by either of the following two "text islands" within a table header:

```
Resp'ble- Party-Action
Respon- sible-
Party-
Action
```
7. If a table header is void—that is, after all special comment characters (including the characters listed in Rule 4) are removed from a table header, there are no remaining valid text islands in the header—and if the table contains a compound statement rather than a function name as part of its specification, the formal parameters will be ordered according to their order of appearance in the initial declarations within the compound statement. A table heading for a table statement that calls a named function may not be void.

The following example is a table header contrived to demonstrate most of the above rules:

**Table** Red-to-3rd-countries

<table>
<thead>
<tr>
<th>country-affected</th>
<th>side cooperation</th>
<th>/=european-==swa-==\</th>
<th>/=involvement==involvement=\</th>
</tr>
</thead>
<tbody>
<tr>
<td>France</td>
<td>Blue</td>
<td>Noncoordinate</td>
<td>Noncombatant</td>
</tr>
<tr>
<td>GDR</td>
<td>Red</td>
<td>Coordinate</td>
<td>On-Call</td>
</tr>
</tbody>
</table>

This example calls the function "Red-to-3rd-countries" twice (once for each row of the table). The data in the table body are matched to five function parameters having the following names: country-affected, side, cooperation, european-involvement, and swa-involvement. Those function parameters need not have been declared or defined in that order.

The use of the vertical bar (|) as a connector character keeping a "text island" together within a *table-header* allows text headings to be associated with individual columns of a table in a very flexible way. Consider the following valid RAND-ABEL table statement:
Table Initialize

<table>
<thead>
<tr>
<th>Country-set [is it a country?, not a region/sea]</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superpower-set [is it one?]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[1-366]</td>
</tr>
<tr>
<td>Player-status [should the model simulate it?]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Borders-WP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[1-366]</td>
</tr>
<tr>
<td>Assertive-country [always fight on attack]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nuclear-capable Mem Orien Red- Blue-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Leader</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Afghanistan</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Arabian-Sea</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Australia</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Austria</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Belgium</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

At times, more table columns are needed to describe a situation than will fit in the width of a single page. To allow wide tables to be described, the following additional format rules for a table-header allow a "wrap-around" header to be created, in which one or more additional rows of "text islands" provide the needed continuation.

Rules for constructing a multi-row table-header:

1. Table headers may be continued onto succeeding lines, if all characters in text islands comprising one row of the header are below all characters comprising the previous row of the header.
2. Within a row of text islands, column headers are read left-to-right.

(Note that by this set of rules, the table above qualifies as having only a single row of column headers, since the vertical bars (|) associating text strings with columns keep the "text islands" defining each column header from being separated vertically.)
Multi-row table headers are best understood by example. Consider the following table statement, with six formal parameters:

<table>
<thead>
<tr>
<th>First-parameter</th>
<th>Second-parameter</th>
<th>Third-parameter</th>
<th>Fourth-parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fifth-parameter</td>
<td>Sixth-parameter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.5</td>
<td>Green</td>
<td>512</td>
<td>&quot;String 1&quot;</td>
</tr>
<tr>
<td>9.0</td>
<td>Blue</td>
<td>(A + 10)</td>
<td>&quot;String 2&quot;</td>
</tr>
<tr>
<td></td>
<td>9943</td>
<td>(A - 24)</td>
<td></td>
</tr>
</tbody>
</table>

Note that a blank line has been used to clearly separate the first and second rows of the table header; this is not strictly necessary, but aids in keeping the text islands separate. Note also that the entries in the table body "wrap around" in the same manner. In fact, the entries are merely read a line at a time, and matched to the corresponding headers in the table header. Although they have been staggered so that they may be placed beneath their corresponding header, this is again not strictly necessary; it simply aids comprehension of the table.

The rules for a table-body are simple: A table-body consists of a sequence of entries, each of which is a unitparam. (See Section IV for the formal definition of a unitparam. It is essentially a primitive value or a parenthesized expression.) The following additional rules hold for a table-body:

Rules:

1. If the table-header describes n formal parameters or variable names, then the number of entries in the table body must be a multiple of n. (Normally, the entries are placed in columns beneath the column headers within the table-header, so that each row of the table naturally consists of n entries, except when wide rows spill over onto the next line as in the example above.)
2. Each entry must match in type with the corresponding formal parameter or local variable.

The table-header and table-body are each followed by a period (.) as delimiter.

The RAND-ABEL Translator that interprets a table merely counts n entries in the table body, either calls the named function or executes the compound-statement, then acquires the next n entries (until a "." is encountered instead). There is no meaning attached to the grouping of table entries into rows.

FUNCTIONS: INVOKING AND EXITING

The declaration of functions was covered in Section V, and Section VI discussed the definition of functions and the syntax for a function-invocation. Functions returning a value are invoked by the expression Report from function-invocation. Functions not returning a value (presumably executed for their side effects) are invoked by a Perform statement, discussed here as part of a description of all RAND-ABEL statements. We also present here the Exit statement that allows completion of a function's execution, whether or not it returns a value.

```
Perform function-invocation.
```

Rule: This statement is used to execute a function which does not return a value. (That is, it is executed for the side effects it causes.)

In some programming languages, a function not returning a value is called a subroutine. In RAND-ABEL, all program logic is contained in functions; a function not returning a value is equivalent to a subroutine.
Example: Perform force-ratio-calc
using France as side-1,
Yugoslavia as side-2,
and 3.5 E 4 as multiplier.

Within the statements defining a function, the following statement is used to return program control to the place from which the function was invoked:

```
statement
Exit.
Exit Reporting simple-expr.
```

Rules:

1. If the Reporting clause is omitted, the function does not return a value.
2. If the Reporting clause is used, the function always returns a value of the same data type as the simple-expr.
3. If there is more than one Exit Reporting statement within the definition of a function, then each of those statements must contain an expression of the same data type. However, the simple-expr may be of an ambiguous data type if that ambiguity is resolved by the function's declaration.

If a function does not return a value, it is always invoked by the RAND-ABEL statement:

```
Perform function-invocation.
```

If a function returns a value, then it is always invoked using the expression:

```
Report from function-invocation
```
Even functions reporting a value may have side effects, and in that sense are not equivalent to a mathematical function.

Examples:

Exit.
Exit Reporting "Success."
Exit Reporting ((multiplier * force-ratio)/2.0).

INPUT/OUTPUT
Explain, Record, Print, Log Statements

The following I/O statements are used to communicate with the "outside world"--the computer system environment within which RAND-ABEL is running.

Before the formal syntax description, some general terms should be understood by the reader. RAND-ABEL operates within a C language environment, within the UNIX operating system. The general characteristics of C and UNIX are assumed. The UNIX system has the (very powerful) concepts of "standard input" (which is often a terminal's keyboard) and "standard output" (which is often a terminal's display screen). Input and output consist of a stream of characters, which are usually directed to the standard input and output ports. However, these data streams can be redirected, for instance into a file, or into the output or input streams of another process running in the computer.

As data are emitted from a RAND-ABEL program, it is either formatted (according to its data type) in a standard (i.e., default) manner, or the programmer can exercise some control over the format in which it appears. A special language of format codes, consisting of a string of characters, is used to specify formatting of I/O. The default formatting for each of RAND-ABEL's data types, and the special format codes, are described in this section.
Statement

Explain unitparam ... unitparam.
Explain with format-spec unitparam ... unitparam.
Record unitparam ... unitparam.
Record with format-spec unitparam ... unitparam.
Print unitparam ... unitparam.
Print with format-spec unitparam ... unitparam.
Print streamname unitparam ... unitparam.
Print streamname with format-spec unitparam ... unitparam.
Log unitparam ... unitparam.
Log with format-spec unitparam ... unitparam.
Log streamname unitparam ... unitparam.
Log streamname with format-spec unitparam ... unitparam.

Rules:

1. The Explain statement and the Log statement work together. The Explain statement appends a stream of data (defined by the sequence of unitparams within the statement) to a special "explain string". It DOES NOT actually perform any output to the log file. Explain only alters the explain string, in preparation for the possibility that a Log statement may be executed sometime later in the program.

2. The Log statement causes the current "explain string" to be sent to the output stream, appending any unitparams included in the Log statement. It has no effect on the explain string, which is "pushed" and "popped" with the stack maintained by an executing C language program; the result is that the explain string has the same scope as a local variable in the program. All entries in the "explain string" caused by Explain statements are lost if a RAND-ABEL program does not issue a subsequent Log statement.
3. The Print statement causes a stream of data (defined by the sequence of unitparams within the statement) to be sent to an output stream.

4. The default output stream is the UNIX stdout; if a streamname is given, output from Print or Log is directed to that output stream instead.\(^1\)

5. The Record statement has the same effect as the Print statement, except that the output stream is a specially designated "record" file.

6. If the optional with format-spec clause is omitted, all output is formatted according to standard defaults determined by the data types of the unitparams being output.

7. If the with format-spec clause is included, the format-spec is a RAND-ABEL expression of type string. The character string is interpreted as a specification for formatting output, and output is formatted according to its specifications.

8. A value of type pointer can be output as a hexadecimal number for debugging purposes, but this is not expected to be used in a production program.

Unitparam is defined in Section IV. Basically, it is a simple value or a parenthesized RAND-ABEL expression.

Default output formats are used for each data type when no control is provided by an explicit format-spec. These default output formats are described in the following subsection.

Examples of the Explain, Record, Print, and Log statements are given at the end of this subsection, after the various formatting options are presented.

---

\(^1\)If the RAND-ABEL program is executing in the context of the RSAC system, output should not be sent to the default stdout, as this will conflict with system CRT screen management.
Default Output Formats

If no special format controls are given, each data type has a standard way in which its value is printed. These default output formats are given by the following table.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>A string of digits, with an optional prefix minus sign. No decimal point. Delimited by one blank on each side.</td>
</tr>
<tr>
<td>Real</td>
<td>A string of digits with an embedded decimal point. At least one digit is printed before and after the decimal point, even if it is a zero. Optional prefix minus sign. Numbers less than one-millionth (1 E -6) are considered zero. Otherwise, for numbers less than one, enough decimal places are printed to show at least two digits of significance. Delimited by one blank on each side.</td>
</tr>
<tr>
<td>String</td>
<td>The string is printed literally, with no surrounding quotation marks, and not delimited by blanks.</td>
</tr>
<tr>
<td>Boolean</td>
<td>The string &quot;Yes&quot; or &quot;No&quot; is printed, without surrounding quotation marks. Delimited by one blank on each side.</td>
</tr>
<tr>
<td>Pointer</td>
<td>A pointer-type value is output as a hexadecimal number for debugging purposes, using the same conventions as an Integer.</td>
</tr>
<tr>
<td>Enumerated</td>
<td>The identifier constant is printed without surrounding quotation marks. Delimited by one blank on each side.</td>
</tr>
</tbody>
</table>

Format Specification

A format-spec is used to control the formatting of output. It is a sequence of characters that are printed as listed, except when the special characters ",", and "\" are encountered. The % is followed by a special formatting code. The formatting codes recognized by RAND-ABEL are as follows:
%i  Enumerated data type
%s  String
%b  Boolean (Yes or No)
%d  Integer without a decimal point
%e  Real (or floating point) scientific notation
%f  Real (or floating point) fixed decimal point
%g  Real (or floating point) general; uses scientific notation
    or fixed decimal point, whichever is shortest.

A literal percent sign is entered in a format string as %%.

A number may be placed between the percent sign and the letter. That number specifies the overall number of characters allocated to the value. If a number is used, the field will be blank padded, unless the field width number begins with a leading zero, in which case the field will be zero padded. The field width number can optionally be followed by a decimal point, and then another number. The second number will be the number of digits to appear after the decimal point for %e, %f, or %g formats.

There are other special options that can be used in these format strings. They obey the conventions of "printf(3)" in Section 3 of the UNIX Programmer's Manual, Bell Laboratories. That document should be consulted for more detailed information.

The backslash (\) is followed by a character or sequence of three octal digits that represent special characters:

\n  Newline (line feed)
\r  Carriage return
\t  Horizontal tab
\b  Backspace
\f  Formfeed
\\  Backslash
\'  Single quote
\\\  Any bit pattern (exactly three digits in octal notation)

These special escape sequences allow any ASCII character to be produced. For example, "\n" allows more than one line of text to be put in the same string, and "\f" causes a page eject. By using a three-digit octal (i.e. base 8 number system) code, any ASCII character can be
produced; e.g., one could make the CRT terminal "bell" ring by the following statement:

\textbf{Print "\007".}

Streams

A stream is a pathway through which information is transferred from a program to a terminal, file, or other program. The information is transferred as a stream of characters.

The normal output stream for \textbf{Log} and \textbf{Print} statements is the UNIX "standard output", which is initially set to the user's terminal.

If a \textit{streamname} has been used in a \textbf{Log} or \textbf{Print} statement, but that stream has not yet been opened, then a run-time error will be generated.

The three pre-declared and pre-opened streams are: "Input", "Output", and "Error".\textsuperscript{2} "Input" is the stream of characters received from the user's terminal keyboard. Reading a character from Input causes UNIX to wait for the user to type in a line of input text. "Output" corresponds to the user's terminal screen. Printing a line on the stream "Output" causes the line to appear on the user's terminal. "Error" is also directed to the user's terminal. It is defined separately from "Output" since the program may want to redefine one of these to go somewhere else.

RAND-ABEL supports three pre-defined stream-oriented functions. They are:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open-stream</td>
<td>file-name</td>
<td>mode</td>
<td>streamname</td>
</tr>
<tr>
<td>Close-stream</td>
<td>streamname</td>
<td></td>
<td>(none)</td>
</tr>
<tr>
<td>Flush-stream</td>
<td>streamname</td>
<td></td>
<td>(none)</td>
</tr>
</tbody>
</table>

\textsuperscript{2}It will help the C programmer to know that these correspond directly to C's \textit{stdin}, \textit{stdout}, and \textit{stderr}.
In the above function calls, \textit{file-name} is a string argument that is either a UNIX file name or a full UNIX pathname (i.e., giving directory, subdirectory, etc.). \textit{Mode} is one of the strings: "read", "write", "append". The value returned from the \textsl{Open-stream} function should be assigned to a integer variable which stores the ID of the stream. This same variable is then used as an argument to the \textsl{Close-stream} and \textsl{Flush-stream} functions.

The \textsl{Open-stream} function associates a UNIX file or path, in read, write, or append mode, with a \textit{streamname}. If a file is opened in write mode and the file does not exist, it is created. If the file does exist, it is deleted first. If a file is opened in append mode, all writing to that file is appended to the end of the existing file, if any.

The \textsl{Close-stream} function closes a stream, making it unavailable for further use (until reopened). It is standard practice to close streams when they will no longer be used by the program.

The \textsl{Flush-stream} function is useful primarily for debugging. Typically, when a RAND-ABEL program executes a \textit{Log} or \textit{Print} statement, the only effect is to fill that file's buffer in the operating system. Later, the operating system will write the file to disk. This buffering of output provides significant performance advantages. The buffer will be written to the appropriate file when a RAND-ABEL program stops execution in the normal manner. However, it is possible for a RAND-ABEL program that has an error to abnormally exit without first writing the buffer to disk. This can cause the programmer to think that his RAND-ABEL program terminated at a point much earlier than is actually the case. To get around this problem, the \textsl{Flush-stream} function can be called to write the contents of the buffer to disk. Because continued use of this function can degrade system performance, it is used primarily for debugging purposes.

Since "Input", "Output", and "Error" are already pre-declared by the system, they can be used to declare other stream variables. For example:

\textbf{Declare} Output-file \textbf{by example}: Let Output-file be Output.
Examples of Input/Output Statements and Functions

The following examples illustrate many of the possible uses of the various input/output statements and functions described in this section:

[ Declare Message-file to be a streamname ]

Declare Message-file by example: Let Message-file be Output.

[ Use the Open-stream function to associate a UNIX file with this streamname, and set its mode to write-only ]

Let Message-file be Report from Open-stream using ":anderson/ABEL/programs/messages" for file-name and "write" for mode.

[ Perform a set of writes to that file ]

Print Message-file with "Threat level is now: %5d in country: %i\n" threat (Report from select-country using Country-list as options).
Print Message-file with "Force ratio is %3.1f at time %d \n" ratio game-time.
Print Message-file "End of game reached. \n"

[ Record program status; note that since a special file is written by the record statement, no file need be opened for this purpose ]

Record "Program reached stage 3.\n"

[ Close file ]


COMPOUND AND NULL STATEMENTS

A compound statement is a sequence of zero or more declarations followed by a sequence of zero or more RAND-ABEL statements, all delimited by braces. It can occur wherever a statement can. It allows more complex program logic to be described than is allowed by the basic set of RAND-ABEL statements:
Rules:

1. All *declarations* occurring within a compound statement are local to that statement; they have no effect outside that statement.
2. Each *declaration* is processed in turn, then each *statement* is executed in turn. To obtain more control flow options, conditional and repetitive execution statements can be used, as well as function invocations.

Notice that a *function-definition* is not allowed within a compound statement. All function definitions are at the "top level" of a RAND-ABEL program.

The compound statement is not terminated by a period, since a period occurs as a delimiter to the last statement within its body. If the compound statement requires more than one line of program text, it is traditional to line up the braces vertically, for ease in visualizing the matches between balancing braces. This positioning is for human consumption only; it is not used by the RAND-ABEL Translator.

Example: The following set of nested conditional statements uses compound statements to denote the set of RAND-ABEL statements to be executed at various places within the conditional logic. This example was given earlier in this manual.
If user-response = "Y" or user-response = "YES"
Then
    { Perform Recalculation.
      Print "Calculation Completed. More? (Y/N): "
      Let user-response be Report from query-user.
    }
Else If user-response = "N" or user-response = "NO"
Then
    { Print "No action taken. More? (Y/N): "
      Let user-response be Report from query-user.
    }
Else If user-response = "?"
    Then Perform Help-function.
    Else
        { Print "Your response not understood."
        Perform Help-function.
    }

A null statement consists of a period, the normal terminating delimiter on a RAND-ABEL statement, only. It is useful within conditional statements to control logic flow.

\[
\text{statement}.
\]

Rule: This statement has no effect.

Note that a side benefit of this statement is that extraneous periods used in error as statement delimiters (for example after a conditional statement) do not cause a syntax error and have no effect.

Example: The following example, used earlier in this manual, illustrates the use of the null statement to control logic flow within a conditional statement.

\[
\text{If king-unchecked Then. Else Perform Think.}
\]
VIII. META-STATEMENTS

The following special statements can be used to influence how RAND-ABEL programs are written and interpreted.

#define

The #Define statement provides an ability to create a macro giving a synonym or alias for a string of characters to be substituted wherever that macro identifier appears:

```
#define name [ unquoted-string ].
```

Rules:

1. The name may be any RAND-ABEL identifier.
2. Wherever that identifier appears, it is replaced by the unquoted-string sequence of characters BEFORE the RAND-ABEL Translator interprets the resulting statement.
3. After replacement, the RAND-ABEL Translator continues its scan at the beginning of the replacement string, so any #Define identifiers it contains will similarly be replaced. #Define statements may be nested to any level.

This form of "macro string substitution" can be used to change the surface appearance of RAND-ABEL programs. It should be used cautiously, since the resulting programs might well become less readable to persons who know the RAND-ABEL language.
Example:

```
#define c-decl
  [ declare country by example:
    let country be {France, Germany, Spain}.
  ].
```

**INCLUDE**

```
meta-statement

Include "filename".
```

**Rules:**

1. The contents of the file whose name (or pathname) is given are inserted at this point in the RAND-ABEL (or C) program. The file name or pathname is interpreted relative to the UNIX directory containing the current source file.

2. After the text insertion takes place, interpretation of the resulting file begins at the start of the newly inserted text lines, so if they contain Include statements, those statements are executed as they are encountered. Include files can be nested up to eight levels deep.

An Include statement is often used to incorporate a standard set of declarations or definitions into a RAND-ABEL program.

Example: Include "libraries/red-agent/dictionary.D".

**MEMORIZE AND RECALL**

```
meta-statement

Memorize name [ unquoted-string ].
Recall name.
```
Rules:

1. The Memorize statement associates a string of text with the given name (any identifier).
2. That text can then be inserted anywhere in the RAND-ABEL program by using the Recall statement.

This pair of statements is useful in "bundling" a set of Data Dictionary declarations under a name, then merely invoking that name wherever they are needed.

Memorize allows a number of program lines to be recalled by a single identifier, and requires the Recall keyword to be used along with the identifier to recall them. By contrast, #Define text cannot include multiple lines, and the identifier itself is sufficient to recall the text. Memorize ... Recall is less efficient than #Define because it creates a temporary file to store the memorized text.

Example:

Memorize my-data
  [ Read: Blue.
    No Read: Red.
    Initialize.
  ].

Recall my-data.

The Recall statement in the example above would be replaced in the RAND-ABEL program by the following Data Dictionary declarations:

Read: Blue.
No Read: Red.
Initialize.
DEBUGGING: TRACE AND UNTRACE

Rules:

1. **Trace** turns on reporting for either **if** statements or function invocations; **Untrace** turns off reporting.
2. All trace data are appended to a special file named "debug.out" within the current UNIX directory.

Function trace data consist of readable statements upon entrance to a function stating the function's name and the values assigned to each of its formal parameters. If the function returns a value, that value is reported to the file upon exit from the function.

"If" trace data write to the same "debug.out" file. Each execution of a conditional statement, when **Trace if** is on, causes the conditional statement itself to be written to the file, along with an indication of whether the **Boolean-expression** evaluated to **Yes** or **No**.

**Trace** and **Untrace** are not executable statements. Rather, they are commands to the RAND-ABEL Translator to embed tracing information within the generated C program. **Trace** and **Untrace** statements can be nested; an **Untrace** turns off the corresponding nearest **Trace** of the same type.

Tracing can significantly reduce the speed of RAND-ABEL program execution, and tends to generate large amounts of output. It should therefore be used selectively and only during program development.
ESCAPE TO C

**Statement**

C Statement @ C-code @.

Rules:

1. The C-code is a sequence of valid C language statements, which is executed according to the normal rules for that language.
2. Anywhere within the C-code that a C-expression can occur, it can be replaced by a RAND-ABEL expression delimited by a back quote mark (\').

The escape back into RAND-ABEL from within C-code is particularly useful in writing valid RAND-ABEL identifiers within the C code, and to permit access to RAND-ABEL data from within the C code.

Although not part of the RAND-ABEL statement syntax, one can also escape to C code wherever a valid RAND-ABEL expression occurs; the RAND-ABEL expression can be replaced by a C language expression enclosed in at-sign (@) marks.

Example:

Declare Cosine by example: Let 1.0 be Report from
Cosine using real-in as x.

Define Cosine:

```
C Statement
@
#include <math.h>
return (double) cos((double) 'x');
@.
```

End.

The above example declares and defines a new RAND-ABEL function that returns the cosine of its argument, called x. It uses the C language math library function "cos". "Cos" takes a C language double-precision floating point argument, and returns a double-precision floating point result. Thus coercion of both input and output of the
C cos function is advisable. The RAND-ABEL variable x is referenced within the C code by means of the "escape back to RAND-ABEL" provided by the back quotes (\'). This function would be called from RAND-ABEL as follows:

Declare cos-answer by example: Let cos-answer be 1.0.
Let cos-answer be Report from cosine using 3.14159 as x.
IX. DATA DICTIONARY

The Data Dictionary facility in RAND-ABEL permits large, complex systems to be developed from separate modules that are created individually by different teams of analysts. It is a much more elaborate and useful facility than the old concept of a "common" area in programs that stores data used in common by the different programs.

The RAND-ABEL Data Dictionary describes the contents and attributes of a data set to be used in common by all the RAND-ABEL modules comprising a system.¹ This common data set contains the specification of:

- A list of items (variables, attributes, tables) to be included in the common data set.
- A list of items, similar to the list above but including sub-procedures and functions, which are not part of the common data set.
- A structuring of the source files which make up the system.
- Access restrictions, ownership, method of implementation, and other such attributes for data items and source code.
- Ancillary information such as the author, module name, and other administrative attributes associated with data items and source code.

The Data Dictionary consists of a set of files that are maintained in an extended RAND-ABEL language. The RAND-ABEL processor translates these files into C language data structures. The resulting C code can be used by a system monitor (a special program providing the foundation for the system being developed) for allocating memory for the data items described, and also by a front-end data editor that can display and manipulate these items.

¹In the Rand Strategy Assessment Center, this common data set is called the World Situation Data Set (WSDS).
A Data Dictionary entry begins with a Declare statement (see Section V). After the declaration of an item follow a number of statements describing the item. The sequence of descriptions is ended when a new Declare is found for the next item, a new Default statement is reached, or the End Declarations statement is encountered.

Many attributes associated with a data item will be the same for an entire group of items (e.g., author, access restrictions, etc.). To avoid the need for typing a whole list of attributes for each item, default attributes may be declared. When a default is declared, it affects all subsequently declared data items within the current file and any files "Included" within that file. A default does NOT affect any files which have "Included" the file which contains it. This nested-default mechanism allows higher-level files to create default environments for lower-level files without worry of a default in a lower-level file causing side effects.

The set of Data Dictionary declarations has the following syntax:

```
data dictionary specification block

Begin Declarations.

declaration

   DDdeclaration . .

   DDdeclaration

   declaration

   DDdeclaration . .

   DDdeclaration

. .

End Declarations.
```

The individual data dictionary declarations (i.e., DDdeclarations) are of three types:
1. **Defining declarations.** Information that actually affects the object code, such as type, size, or access data.

2. **Identifying declarations.** Information that is documentary but mandatory.

3. **Informative declarations.** Information that is optional but useful as documentation.

Each of these categories of declarations is described below.

**DEFINING DECLARATIONS**

These declarations are mandatory for each external data item, but may be handled by default declarations that are in effect (see p. 81).

```
- DDdeclaration -

Method: Direct.
Function.
Macro.

Function: func-name.
Macro: string-literal.
```

Rule: **Method** means "method of access". An item's access method tells whether the variable is accessed directly or via a function or macro. If an item is accessed via a macro, the macro must be defined using a **Macro** statement. If it is accessed via a function, the name of the function must be given using the **Function** statement. The **Macro** or **Function** statement must immediately follow the **Method** declaration.

Examples: **Method Function.**

```
Function: calculate-attrition.
```

**Method Macro.**

```
Macro: gov2cntry(F,i)
   (*((GOVERN_entry)(char *)F + i) - F->governs + 1.
```
Rule: The **Use** declaration indicates whether an item is to be created dynamically (**Clone**) when the Push function is invoked and discarded when the corresponding Pop function is executed; or whether one instance of the data item is to be maintained throughout a Push and Pop (**No Clone**). Data that are never changed during program execution are declared with the **Constant** option.

Example: **Use: No Clone.**

Rule: This declaration allows different modules to have separate items with the same name. Source code also has an owner, and automatically accesses either its own or "global" data items, unless otherwise specified by this declaration. The "**owner-name**" is one of a set of commonly agreed-upon names by which the various groups developing code are identified. The keyword **Global** is used if there is no specific owner.

Examples: **Owner: Red.**

**Global.**
**Rule:** These declarations specify which source code owners (i.e. "access groups") can read and/or write this item. The "No" prefix can turn off a default or serve documentary purposes by establishing a lack of access for a particular group. The special group *Everyone* applies to all access groups, and can be used to grant or deny access for all groups.

**Examples:**

- **Noread** Blue, Neutral.
- **Write Everyone.**

---

**Rule:** The preferred format for reading and writing this data item is stated as a quoted string of format descriptors (see Section VII).²

²For programmers familiar with the C language: the format
It is desirable to specify output formats for string variables, integers, and real (floating point) data whenever possible, since it helps the display programs format the data in a consistent manner. It is unnecessary to specify output formats for enumerated variables since the field width needed to display them is easily determined by the display programs.

Example: Write Format: "%.3f".

**DDdeclaration**

Validation Range: numeral to numeral.

Validation Function: func-name.

Rule: In the first form, an inclusive range of numeric values (either integer or real) is given. In the second form, a Boolean function is named that is expected to return Yes for a valid item, and No otherwise.

Examples: Validation Range: 2.7 TO 8.7;.
Validation Function: Check-value.

**DDdeclaration**

Prompt Function: func-name.

String: string-literal.

Rule: In the first form, the named function will be called prior to input for this data item. It returns a string which will be displayed on the user's CRT screen. In the second form, a quoted character string is given for display prior to input of the data item.

specification is the same as those used for the scanf and printf functions.
If a RAND-ABEL simulation is being performed under the control of the system monitor, the monitor passes any string generated by the Prompt declaration to the user's terminal.

Examples: **Prompt Function:** Show-message.  
**Prompt String:** "Type ratio as a decimal: ".

**DDdeclaration**

- Initialize.
- No Initialize.

Rule: This declaration indicates whether the item can be initialized by the RAND-ABEL Translator or not.

At this time, the **Initialize** declarations may be used in a program, but their effect has not been implemented. Consequently, they do not alter program behavior.

Examples:  
- Initialize.
- No Initialize.

**IDENTIFYING DECLARATIONS**

These declarations are mandatory, but take an arbitrary comment as an argument. They are used for standardized documentation of a module.
Rule: The comment may be a free-format comment that aids in the documentation of the program or data item.

Examples: Author: [ Mark LaCasse ]. Date: [ 83/02/05 ]. Definition: [ This function returns a string value which should be displayed prior to input of the force structure ratio ].

INFORMATIVE DECLARATIONS

Informative declarations are optional. They provide additional structured documentation of a RAND-ABEL program module.

Rule: The comment may be a free-format comment that aids in the documentation of the program or data item.

The Status declaration is often used to represent whether a variable is "proposed" (indicating the author is willing to entertain proposals for change), or "confirmed" (indicating the author has closed debate on the subject).
Examples:

References: [ See R-1258, Section II ].
Comments: [ This function is a placeholder until a more complete computation can be developed ].
Status: [ Proposed ].

CREATING AND REMOVING DEFAULT DECLARATIONS

As mentioned above, all the mandatory Data Dictionary declarations need not be given for each data item or function. Many of these declarations can be covered by use of declared defaults.

Any of the DDdeclarations described in this section may be preceded by the keyword Default. If that is done, that setting for the particular DDdeclaration remains in force within the current file (and files included within it) until a new default is given or a No Default is declared for that type of DDdeclaration. Any default setting may be overridden by individual DDdeclarations associated with a particular data item or function.

Examples:

Default Owner: Red-Agent.
Default Method: direct.
Default No Initialize.
Default Author: [ Mark LaCasse, randvax!lacasse ].

The following No Default statements may be issued to remove a default setting on a type of DDdeclaration:
When No Default is specified for any mandatory DDdeclarations, such a DDdeclaration must accompany each data item or function until the next Default or End Declarations statement is reached.

Examples of No Default statements:

No Default Comments.
No Default Prompt String.
No Default Read Format.
No Default Validation Function.

EXAMPLE OF A DATA DICTIONARY DECLARATION SECTION

The following is a complete example of a Data Dictionary declaration section within a RAND-ABEL program. Such a RAND-ABEL code section is often contained in a file that can be included within another RAND-ABEL file to obtain the standard defaults and definitions required.
[ Sample Data Dictionary Entries for Blue Agent, January 1984 ]

Begin declarations.

Default Owner: Blue.
Default Method: Direct.
Default Read: Blue.
Default Write: Blue.
Default Use: Clone.
Default No Initialize.
Default Author: [ Mark LaCasse, randvax!lacasse ].
Default Date: [ 84/01/05 ].

Declare Lookahead-opponent by example:
Let Lookahead-opponent be
{
    BR1, [ Blue's Red version one ]
    BR2 [ Blue's Red version two ]
}.
Definition: [ Blue's Red, opposes Blue in Lookaheads ].

Declare Time-to-wake by example:
Let Time-to-wake be 45786.
Prompt String: "Enter the date and time in the format: MMM DD, hh.mm".
Validation Function: Check-time-input.
Definition: [ General purpose, future time to wake Blue ].

End Declarations.
X. CO-PROCESSES

A co-process is a process (that is, an executing program) that is started by another process, which then executes independently and asynchronously of its parent process. This section discusses how co-processes are created, put to sleep, and terminated.

CREATING A CO-PROCESS

Co-processes are created by calling a built-in RAND-ABEL function called "Start-co-process". It takes two arguments, proc-start and proc-name. Proc-start is given an object representing the top-level function of the to-be-created process. This object can be created by use of the function expression. Proc-name is a string identifying the new process. The function Start-co-process returns an object which is of type process. Therefore, the initiation of a new process might be performed by a statement such as the following:

Declare new-proc by example: Let new-proc be Self.

Let new-proc be Report from Start-co-process using (Function Top-of-proc) as proc-start and "Offensive strategy" as proc-name.

Note the use of the expression (Function Top-of-proc) to create an object representing the user's function named "Top-of-proc", which is to be executed as the beginning of the new process.

The execution of the Start-co-process function is the only method by which a value of type process can be created.

PUTTING A PROCESS TO SLEEP

A process can cause itself to "go to sleep"--that is, to stop processing until it is awakened by some external program. This is done by activating a built-in function called "Sleep". Sleep takes no arguments and returns no value:

Perform Sleep.
There are no facilities within the RAND-ABEL language itself for awakening a function once it is sleeping; at present, those facilities are part of the support environment in which a RAND-ABEL process resides, and must be invoked directly within that support environment. (See Appendix A for some further information on the support environment for RAND-ABEL.)

**TERMINATING A CO-PROCESS**

To terminate a co-process, the "Kill-co-process" function is used. It takes one argument, called process: an object of type process that identifies the process to be terminated. Therefore, to terminate the process created by the example above, one would write:

```
Perform Kill-co-process using new-proc as process.
```

**RESERVED CO-PROCESS VARIABLES: SELF AND PARENT**

The RAND-ABEL system contains two reserved process-type variables: Self and Parent. Self is always equal to the current process. Parent always refers to the parent process that spawned a given process. These variables can also be used in Declare statements as examples of processes, so that new variables of type process can be declared.

**RULES FOR THE USE OF CO-PROCESSES**

The following rules govern the use of co-processes.

1. A co-process may not call Kill-co-process on itself.
2. Processes can be the object of assignment statements, so that statements such as:

   `Let me be Self.`

   are valid.
3. Processes can be values of suitably defined variables or arrays. This was illustrated in the above example, where "me" is a variable taking on a process as its value.
4. Processes can be arguments of suitably declared arrays. For example:

\begin{verbatim}
Declare active_process by example:
    Let active_process be Self.

Declare array by example:
    Let array of active_process be 17.
\end{verbatim}

5. Processes can be parameters of functions. For example, the built-in function "Kill-process" has as its single parameter an object of type process.

6. Processes are subject to no other operations.
XI. TOP-LEVEL RAND-ABEL DECLARATIONS, DEFINITIONS, AND STATEMENTS

The RAND-ABEL "language" is a complete programming language. However, only certain of the RAND-ABEL statements and declarations can occur at the top level of a RAND-ABEL program. All other RAND-ABEL constructions occur within these top-level statements and declarations.

The only RAND-ABEL declarations and statements that can occur at the top level are:

- Any valid RAND-ABEL Declare
- Any valid RAND-ABEL Define
- The RAND-ABEL statements:

  C Statement @ C-code @

  Trace If.
  Function.

  Untrace If.
  Function.

- One of the two mutually exclusive statements:

  Owner: name.
  
  Global.

- A special top-level-only Declare:

  Declare Ignore name . . . name.
  name, . . . , name.

- The set of declarations providing information to the Data Dictionary facility (see Section IX).
Begin Declarations.

declaration

DDdeclaration ... DDdeclaration

declaration

DDdeclaration ... DDdeclaration

End Declarations.

The meanings of all normal RAND-ABEL Declares, Defines, and statements are found in earlier sections of this Note. The meaning of the ownership statements (Owner and Global) is found in Section IX; when used here, the statements declare the ownership tag to be put on all subsequent RAND-ABEL code.

The Declare Ignore statement is used to add a set of identifiers to a list (initially null) that the RAND-ABEL Translator will ignore whenever they are encountered. These identifiers can then be used as "noise words" in RAND-ABEL statements, presumably to increase their readability.

For example, the declaration:

Declare ignore a, an, the.

allows one to write a RAND-ABEL statement such as:

Let the color of a piece be white.

which is equivalent to the RAND-ABEL statement using explicit comments:

Let [the] color of [a] piece be white.

or, the more brutally simple:

Let color of piece be white.

Care should be taken in the declaration and use of such noise words, since readers of RAND-ABEL code might overlook the Declare Ignore statement and believe that these words are part of valid
RAND-ABEL syntax, possibly leading them to write incorrect RAND-ABEL programs. Also, it should be noted that in normal English the phrases "a piece" and "the piece" mean quite different things, whereas they do not in a RAND-ABEL program in which both "a" and "the" are declared to be noise words, again creating the possibility of confusion in the RAND-ABEL reader's mind.
Appendix A
LOCAL SUPPORT ENVIRONMENT FOR RAND-ABEL

Appendix A briefly discusses the use of RAND-ABEL at The Rand Corporation. Its contents are specific to this site. Currently, RAND-ABEL is used at Rand only in the context of the Rand Strategy Assessment Center (RSAC). Consequently, the following guidelines are RSAC-specific.

The RAND-ABEL Translator is generally used in two modes: (1) as an aid in preparing syntactically correct RAND-ABEL rules, and (2) to produce compilable C code for actual incorporation into the executable RSAC model. The basic difference between (1) and (2) is the handling of the Data Dictionary.

Essentially, in preparing RAND-ABEL programs data items are frequently added, changed, or removed; as a result, the writer cannot use the master Data Dictionary, but his own extract of it. This seeming inconvenience can actually be an aid, as it forces the writer to be aware of how his program integrates with the other parts of the model.

In all cases, the writer incorporates the Data Dictionary into the program by the ABEL Include statement (described in Section VIII). Since this mechanism allows nesting (that is, included files can contain other include statements), a two-level approach is used. At the top of a RAND-ABEL program file is an Include for a single Data Dictionary file; this file in turn contains Includes for all needed Data Dictionary components. In the finished program, the program file instead incorporates the master Data Dictionary file.

All Data Dictionary files end in ".D", whereas all RAND-ABEL program files end in ".A". The RAND-ABEL Translator is applied to the ".A" files only (with the Includes introducing the Data Dictionary). The result of applying the RAND-ABEL Translator is (possibly) a series of error messages and two files: a ".A.c" file and a ".A.I" file. These files are then used to build the integrated RSAC model.
All Data Dictionary and RAND-ABEL program files are eventually registered with the RSAC Data Dictionary administrator for the final integration.

Running the RAND-ABEL Translator itself is quite simple. Applying it to the file "rules.A" would involve typing (on the randvax or text processor (TP) computers1 used by RSAC):

```
/s/sl/rsac/bin/enabel rules.A
```

Most people will probably want to use an alias for this, and so will place the line:

```
alias enabel /s/sl/rsac/bin/enabel
```

into their ".cshrc" files.

In reviewing the error messages produced by the RAND-ABEL Translator, two things should be kept in mind:

- The range of legal RAND-ABEL language inputs is quite large; sometimes it takes several keywords for the RAND-ABEL Translator to detect that an error has occurred. Thus, not only should the line number and word printed by the RAND-ABEL Translator be checked for the error, but the code immediately preceding it as well.

- A problem with block-structured languages (like RAND-ABEL) in general, and with top-down parsers like the RAND-ABEL Translator in particular, is a difficulty in recovering from certain errors. (That is, there is often a problem in finding the start of legal statements after the error, and in recovering context skipped over because of the error.) As a result, it is quite possible for a single error to produce a hundred error messages. Thus, there are occasions where only

---

1The randvax and TP computers are both VAX 11/780s running 4.2BSD UNIX. The TP computer is authorized to handle material at the SECRET level.
the first error reported is an actual problem, while subsequent error messages are a result of declarations, definitions, or statement boundaries having been missed.

Note that this is not always the case. If a several-line gap appears between errors, there is a good chance that the later error is valid. However, an error in a declaration can cause spurious errors wherever the item being declared is subsequently used, even if thousands of error-free lines intervene.

In all cases, the RAND-ABEL program writer has to learn through experience and through sharing experience with other writers. Also, there are certainly cases where the RAND-ABEL Translator can do a better job of finding and reporting errors, and in recovering from them. Communication is thus very important.
Appendix B
QUICK-REFERENCE GUIDE TO THE RAND-ABEL LANGUAGE

The following is a list of RAND-ABEL keywords. Words that always occur in sequence as phrases are shown together; words are shown separately that are optional or are one of several possible choices within a phrase.

<table>
<thead>
<tr>
<th>Address of and are not as Attribute Author</th>
<th>For Format from Function Global</th>
<th>Parent Perform plus Print Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin Declarations Break by Clone Comments</td>
<td>Concatenated with Constant Continue</td>
<td>C Statement</td>
</tr>
<tr>
<td>Date Declare...by example Decrease by Default Define</td>
<td>Define</td>
<td>#Define</td>
</tr>
<tr>
<td>End End Declarations Erasable Erase Evaluate Everyone Exit Explain</td>
<td>Explain</td>
<td>Of Occupant of or Owner</td>
</tr>
<tr>
<td>For Format from Function Global</td>
<td>Parent Perform plus Print Prompt</td>
<td></td>
</tr>
<tr>
<td>Begin Declarations Break by Clone Comments</td>
<td>Concatenated with Constant Continue</td>
<td>C Statement</td>
</tr>
<tr>
<td>Date Declare...by example Decrease by Default Define</td>
<td>Define</td>
<td>#Define</td>
</tr>
<tr>
<td>End End Declarations Erasable Erase Evaluate Everyone Exit Explain</td>
<td>Explain</td>
<td>Of Occupant of or Owner</td>
</tr>
<tr>
<td>For Format from Function Global</td>
<td>Parent Perform plus Print Prompt</td>
<td></td>
</tr>
<tr>
<td>Begin Declarations Break by Clone Comments</td>
<td>Concatenated with Constant Continue</td>
<td>C Statement</td>
</tr>
<tr>
<td>Date Declare...by example Decrease by Default Define</td>
<td>Define</td>
<td>#Define</td>
</tr>
<tr>
<td>End End Declarations Erasable Erase Evaluate Everyone Exit Explain</td>
<td>Explain</td>
<td>Of Occupant of or Owner</td>
</tr>
<tr>
<td>For Format from Function Global</td>
<td>Parent Perform plus Print Prompt</td>
<td></td>
</tr>
<tr>
<td>Begin Declarations Break by Clone Comments</td>
<td>Concatenated with Constant Continue</td>
<td>C Statement</td>
</tr>
<tr>
<td>Date Declare...by example Decrease by Default Define</td>
<td>Define</td>
<td>#Define</td>
</tr>
<tr>
<td>End End Declarations Erasable Erase Evaluate Everyone Exit Explain</td>
<td>Explain</td>
<td>Of Occupant of or Owner</td>
</tr>
<tr>
<td>For Format from Function Global</td>
<td>Parent Perform plus Print Prompt</td>
<td></td>
</tr>
<tr>
<td>Begin Declarations Break by Clone Comments</td>
<td>Concatenated with Constant Continue</td>
<td>C Statement</td>
</tr>
<tr>
<td>Date Declare...by example Decrease by Default Define</td>
<td>Define</td>
<td>#Define</td>
</tr>
<tr>
<td>End End Declarations Erasable Erase Evaluate Everyone Exit Explain</td>
<td>Explain</td>
<td>Of Occupant of or Owner</td>
</tr>
</tbody>
</table>
In addition, the following special symbols also act as keywords having special meaning:

@   --   (...)  (...)  [...]  ,  :  
+   -   *   /   %   ~   $  
=   ~=   >=   <=   >   <   æ   |  

In the above lists, ellipses (...) represent intervening words in a standard phrase. (Elsewhere in this manual, ellipses represent certain syntactic options.)

The following pages contain a summary of the RAND-ABEL syntax charts contained within this Note.

expression

Report from function-invocation

Evaluate unitparam . . . unitparam
   with format-spec unitparam . . . unitparam

   numeral variable-name
   unary-operator expression
   expression binary-operator expression

There is array-access
   a array-access
   an array-access

Describe array-name

simple-expr
\[ \text{simple-expr} \]

Function \( \text{function-ptr} \)
\( \text{function-name} \)

unitparam

Yes
No
\@ c-code @
enumerated-value
numeric-literal
quoted-string
variable-name
Unspecified
--
( expression )

lvalue

variable-name

Occupant of \( \text{variable-name} \)
array-access

Address of \( \text{variable-name} \)
Function \( \text{function-name} \)
Attribute \( \text{array-name} \)

array-access

\[ \text{simple-expr} \text{array-name} \text{simple-expr} \]

array-name of \( \text{simple-expr} \),
\( \text{simple-expr} \),
\( \text{simple-expr} \),
in and
by and
and
and
and
Declare \textit{variable-name} by example:

Let \textit{variable-name} be \textit{expression}.

\{ \textit{name} \ldots \textit{name} \}.
\{ \textit{name}, \ldots, \textit{name} \}.

Declare \textit{array-name} by example:

Let \textit{array-name} of \textit{simple-expr}, \ldots.

\textit{in} \quad \text{and} \quad \textit{by} \quad \text{and}

\ldots, \quad \textit{simple-expr} \quad \text{and} \quad \text{and}

\textit{be} \textit{expression}.

\{ \textit{name} \ldots \textit{name} \}.
\{ \textit{name}, \ldots, \textit{name} \}.

Declare \textit{array-name} by example:

Let \textit{simple-expr \ array-name \ simple-expr}

\textit{be} \textit{expression}.

\{ \textit{name} \ldots \textit{name} \}.
\{ \textit{name}, \ldots, \textit{name} \}.

Declare \textit{func-name} by example:

Let \textit{expression} be \textit{Report from} \textit{named-function-call}.
Perform \textit{named-function-call}.
function-definition
Define func-name : declaration . . .
  declaration
  statement . . .
  statement
End.

named-function-call
  func-name
  func-name using expression as param-name , . . .
    for and
    , and
    . . . , expression as param-name
    and for
    , and

function-invocation
  named-function-call
  func-ptr
  func-ptr using expression as param-name , . . .
    for and
    , and
    . . . , expression as param-name
    and for
    , and
statement

Let \( \text{Ivalue} \) be \( \text{expression} \).
pointer

Increase \( \text{Ivalue} \) by \( \text{expression} \).
Decrease
Multiply
Divide

If \( \text{Boolean-expression} \) Then \( \text{statement} \)

If \( \text{Boolean-expression} \) Then \( \text{statement} \) Else \( \text{statement} \)

For \( \text{variable} : \text{statement} \)

While \( \text{Boolean-expression} : \text{statement} \)

Continue.

Break.

Table \( \text{func-name} \)
    \( \text{compound-statement} \)
        \( \text{table-header} \).
        \( \text{table-body} \).

Perform \( \text{function-invocation} \).

Exit.
Exit Reporting \( \text{simple-expr} \).

Explain \( \text{unitparam} \ldots \text{unitparam} \).
Explain with \( \text{format-spec unitparam} \ldots \text{unitparam} \).

Record \( \text{unitparam} \ldots \text{unitparam} \).
Record with \( \text{format-spec unitparam} \ldots \text{unitparam} \).

Print \( \text{unitparam} \ldots \text{unitparam} \).
Print with \( \text{format-spec unitparam} \ldots \text{unitparam} \).

Print \( \text{streamname unitparam} \ldots \text{unitparam} \).
Print \( \text{streamname with format-spec unitparam} \ldots \).

    \( \ldots \text{unitparam} \).

Log \( \text{unitparam} \ldots \text{unitparam} \).
Log with \( \text{format-spec unitparam} \ldots \text{unitparam} \).
Log streamname unitparam . . . unitparam.
Log streamname with format-spec unitparam . . .

. . . unitparam.

{ declaration . . . declaration
    statement . . . statement }

Trace If.
    Function.
Untrace If.
    Function.
C Statement @ C-code @.
data dictionary specification block

Begin Declarations.

declaration

DDdeclaration . . .

DDdeclaration

declaration

DDdeclaration . . .

DDdeclaration

...

End Declarations.

DDdeclaration

Method: Direct.
Function.
Macro.

Function: func-name.

Macro: string-literal.

Use: Clone.
No Clone.
Constant.

Owner: owner-name.

Global.

Read Everyone.

owner-name . . . owner-name.
owner-name , . . . , owner-name.

Noread Everyone.

owner-name . . . owner-name.
owner-name , . . . , owner-name.

Write Everyone.

owner-name . . . owner-name.
owner-name , . . . , owner-name.
Nowrite Everyone.

\[ \text{owner-name} \ldots \text{owner-name} \]
\[ \text{owner-name}, \ldots, \text{owner-name} \]

Read Format: string-literal.
Write Format: string-literal.
Validation Range: numeral to numeral.
Validation Function: \text{func-name}.
Prompt Function: \text{func-name}.
String: string-literal.

Initialize.
No Initialize.
Author: \text{comment}.
Date: \text{comment}.
Definition: \text{comment}.
References: \text{comment}.
Comments: \text{comment}.
Status: \text{comment}.
No Default Author.
Comments.
Date.
Definition.
Initialize.
Method.
Owner.
Prompt Function.
String.
Read Format.
Write Format.
References.
Status.
Use.
Validation Function.
Range.
meta-statement

#define name [ unquoted-string ].
#include "filename".

memorize name [ unquoted-string ].
recall name.
INDEX

#Define meta-statement 67
$ string concatenation operator 26
* codes, used for I/O formatting 61
& logical and operator 25
* (Multiplication operator) 21
+ (Addition operator) 21
, and 19
-- (synonym for Unspecified) 13
- (Subtraction operator) 21
- (Unary negation sign) 21
... notation, meaning of 5
A RAND-ABEL program files at Rand 91
.A.c file, produced by RAND-ABEL Translator 91
.A.I file, produced by RAND-ABEL Translator 91
.cshrc file, inclusion of an alias within 92
.D data dictionary files at Rand 91
/ (Division operator) 21
< operator 23
<= operator 23
= operator 22
>= operator 23
> operator 23
@ symbol used in escape to C code 71
@ c-code @ expression 19
\ (backslash) codes, used in format-spec for I/O 61
' (back quote) used to escape to RAND-ABEL expression from C code 71
{...} used to delimit compound statement 65
| logical or operator 25
~= operator 22
~ logical not operator 25

Abbreviation of parameter names in table-header 51
Addition operator (+) 21
Address of phrase 19
Alias for RAND-ABEL Translator access path, how to create 92
Aliases, creation of using #Define meta-statement 67
and (&) logical operator 25
and 19
an (in There is expression) 16
an 34
are not operator 22
Array declaration, infix form of 30
Array declaration, prefix form of 30
Array Declarations 29
array-access 15
array-access, definition of 19
ASCII characters, production of in output 61
Assignment statement 40
as 37
as 38
Author data dictionary declaration 80
a (in There is expression) 16
a 34

Back quote (') used to escape to RAND-ABEL expression from C code 71
Backslash (\) codes, used in format-spec for I/O 61
Begin Declarations 74
binary-operator 16
Boldface, meaning of 4
Boolean data type, default output format for 60
Boolean values, result of logical operators 25
Boolean, built-in data type 12
Break statement 45
Buffers in RAND-ABEL I/O 63
Built-in RAND-ABEL data types 12
by 19

C code within a RAND-ABEL program 71
C programming language 1
C programming language 2
C statement 71
Case shifts, in RAND-ABEL keywords 5
Case shifts, unique in identifiers 7
Character string, built-in data type 12
Clone option in Use declaration 76
Close-stream function 62
Co-processes 84
Co-processes, rules for the use of 85
Co-routines (see co-processes) 84
Comments, notation for 9
Comments data dictionary declaration 80
Comparison of two enumerated values 24
Comparison of two strings 24
Comparison Operators 22
Compound statement 65
concatenated with string operator 26
Conditional execution 42
Constant option in Use declaration 76
Continue statement 45
Creating a co-process using Start-co-process function 84
Creating a data value of type process 84

Data dictionary declaration section, complete example 83
Data dictionary files at Rand 91
data dictionary specification block 74
Data Dictionary 73
Data types 12
Date data dictionary declaration 80
Debug.out file 70
Debugging, statements useful for 70
Decision table, example 49
Declaration of functions 34
Declarations 28
Declare ignore statement 88
Declare...by example array declaration 30
Declare...by example declaration 28
Decrease...by statement 41
Default declarations, creating and removing 81
Default output formats 60
Default data dictionary declaration 81
Define statement for function definition 36
Defining declarations within a data dictionary declaration 75
Definition data dictionary declaration 80
Describe expression 16
direct option in method declaration 75
Divide...by statement 41
divided by operator 21
Division by zero 22
Division operator (/) 21

Ellipsis (...), meaning of 5
Else 42
enable, program name of RAND-ABEL Translator 92
End Declarations 74
End 36
Enumerated data type 13
Enumerated data type, default output format for 60
Enumerated data types, creation of 28
Enumerated variable 13
Equality test for character strings 23
Equality test for enumerated values 23
Equality Tests 22
Erasable Arrays 31
Erasable 32
Erase statement for arrays 33
Error messages produced by RAND-ABEL Translator 92
Error stream 62
Escaping to C code 71
Evaluate...with expression 16
Evaluate expression 16
everyone option in read and write declarations 77
Existence of array elements, testing for 34
Exit reporting 36
Exit statement 56
Explain string, used by Log statement 58
Explain statement 58
Exponential notation 12
expression, definition of 16
expression, meaning of 15

File read/write access data dictionary declarations 77
Flush-stream function 62
format-spec  16
format-spec  58
format-spec, definition of  60
Formats, default output  60
For statement  44
Function Definition  36
function-invocation  38
Functions, declaration of  34
Functions, invoking and exiting  55
Function expression  19
function option in method declaration  75

Global data dictionary declaration  76

Hyphen (-) and underscore (_), in identifiers  7

Identifier constant  13
Identifier constant, default output format for  60
identifier  7
Identifying declarations  79
If...Then...Else statement  42
Implied multiplication  17
Include meta-statement  68
Increase...by statement  41
Inequality of two identifier constants  24
Inequality of two strings  24
Inequality Tests  23
Infix form of array declaration  30
Infix manner of accessing arrays  20
Information declarations  80
Initialize data dictionary declaration  79
Input stream  62
Input/output statements  57
Integer data type, default output format for  60
Integer, built-in data type  12
Invoking a function  55
in  19
is at least operator  23
is at most operator  23
is greater than operator  23
is less than operator  23
is not operator  22
is operator  22
Italics, meaning of  5

Keywords, RAND-ABEL  4
Keywords, table of all RAND-ABEL  94
Kill-co-process function to terminate a process  85

Let...be statement  41
List structures, for storage of sparse arrays  32
Local support environment for RAND-ABEL  91
Logical data type 12
Logical Operators 25
Logical operators, meaning of 25
Log statement 58
lvalue 15
lvalue, definition of 19

Macro definitions, using #Define meta-statement 67
macro option in method declaration 75
Make...Erasable statement 32
Make...Semi-erasable statement 32
Make...Unerasable statement 32
Matrix, represented in RAND-ABEL 29
Memorize meta-statement 68
Meta-statements 67
Method data dictionary declaration 75
minus operator 21
modulo operator 21
Multiple rows in table-header 53
Multiplication operator (*) 21
Multiplication, implied 17
Multiply...by statement 41

named-function-call, definition of 37
negative operator 21
No clone option in Use declaration 76
No Default data dictionary declaration 82
No Initialize data dictionary declaration 79
Noise words, ability to declare and use 88
Noread data dictionary declaration 77
not (~) logical operator 25
Nowrite data dictionary declaration 77
No 12
No 19
Null statement 66
Numeric Operators 21

Occupant of phrase 19
of 19
Open-stream function 62
Options, notation for 5
or (|) logical operator 25
Output formats, default 60
Output stream 62
Owner data dictionary declaration 76

Parenthesized expression 19
Parent reserved co-process variable 85
Parent 12
Perform statement 38
Perform statement 55
plus operator 21
Pointer data type, default output format for 60
Precedence relations for RAND-ABEL operators 27
Prefix form of array declaration 30
Prefix manner of accessing arrays 21
Print statement 58
Process, built-in data type 12
Prompt function/string data dictionary declaration 78
Quick-reference guide to RAND-ABEL syntax 94
Rand Strategy Assessment Center (RSAC) 1
Rand Strategy Assessment Center (RSAC) 91
RAND-ABEL keywords 4
RAND-ABEL local support environment 91
RAND-ABEL syntax, quick-reference guide to 94
RAND-ABEL Translator 1
RAND-ABEL Translator 2
RAND-ABEL Translator, access to at Rand 92
Read format data dictionary declaration 77
Read data dictionary declaration 77
Real data type, default output format for 60
Real, built-in data type 12
Recall meta-statement 68
Record statement 58
References data dictionary declaration 80
Repetitive execution 43
Report from expression 16
Report from expression 38
Reporting 56
Returning control from a function with exit statement 56
ROSIE programming language 2
RSAC (Rand Strategy Assessment Center) 1
RSAC, Rand Strategy Assessment Center 91
Self reserved co-process variable 85
Self 12
Semi-erasable 32
simple-expr 15
simple-expr, definition of 19
Sleep function for co-processes 84
Sparse Arrays 31
Special characters within a table-header 51
Special symbols, table of all RAND-ABEL 95
Standard input 57
Standard output 57
Start-co-process function 84
statement 40
Status data dictionary declaration 80
stderr 62
stdin 62
stdout as standard output stream 59
stdout  62
streamname  58
Streams (for I/O)  62
String data type, default output format for  60
String Operator  26
String, built-in data type  12
Strongly typed language  11
Subroutines, same as functions in RAND-ABEL  55
Subtraction operator (-)  21
Support environment for RAND-ABEL  91
Synonyms, creation of using #Define meta-statement  67
Syntax of RAND-ABEL, quick-reference guide to  94

table-body, rules for constructing  54
table-header, rules for constructing  50
Table statement  46
Terminating a co-process  85
Text island, within a table-header  50
Then  42
There is expression  16
There is expression  34
times operator  21
Top-level RAND-ABEL declarations, definitions, and statements  87
Trace statement  70
Truncation of result in integer division  21

unary-operator  16
Underscore (_) and hyphen (-), in identifiers  7
Unerasable  32
unitparam  15
unitparam, definition of  19
UNIX  1
Unspecified  13
Unspecified  19
Untrace statement  70
Use Clone/No Clone/Constant data dictionary declaration  76
using  37
using  38

Validation range/function data dictionary declaration  78
Vector, represented in RAND-ABEL  29

While statement  44
White space, in RAND-ABEL statements  9
World Situation Data Set (WSDS)  73
Write format data dictionary declaration  77
Write data dictionary declaration  77

Yes  12
Yes  19
REFERENCES


