Rational Environment Reference Manual

String Tools (ST)

Document Control Number: 8001A-28

Rev. 1.0, November 1985
Rev. 2.0, April 1986
Rev. 2.1, July 1986
Rev. 3.0, July 1987 (Delta)

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

How to Use This Book

The String Tools (ST) book of the *Rational Environment Reference Manual* contains reference information describing some of the tools for programmatic manipulation of strings provided by the Rational Environment™. It is intended for users who are familiar with the Environment and with Ada® programming. Note that the reference information describing some of the programming tools is documented in Programming Tools (PT) of the *Rational Environment Reference Manual.*

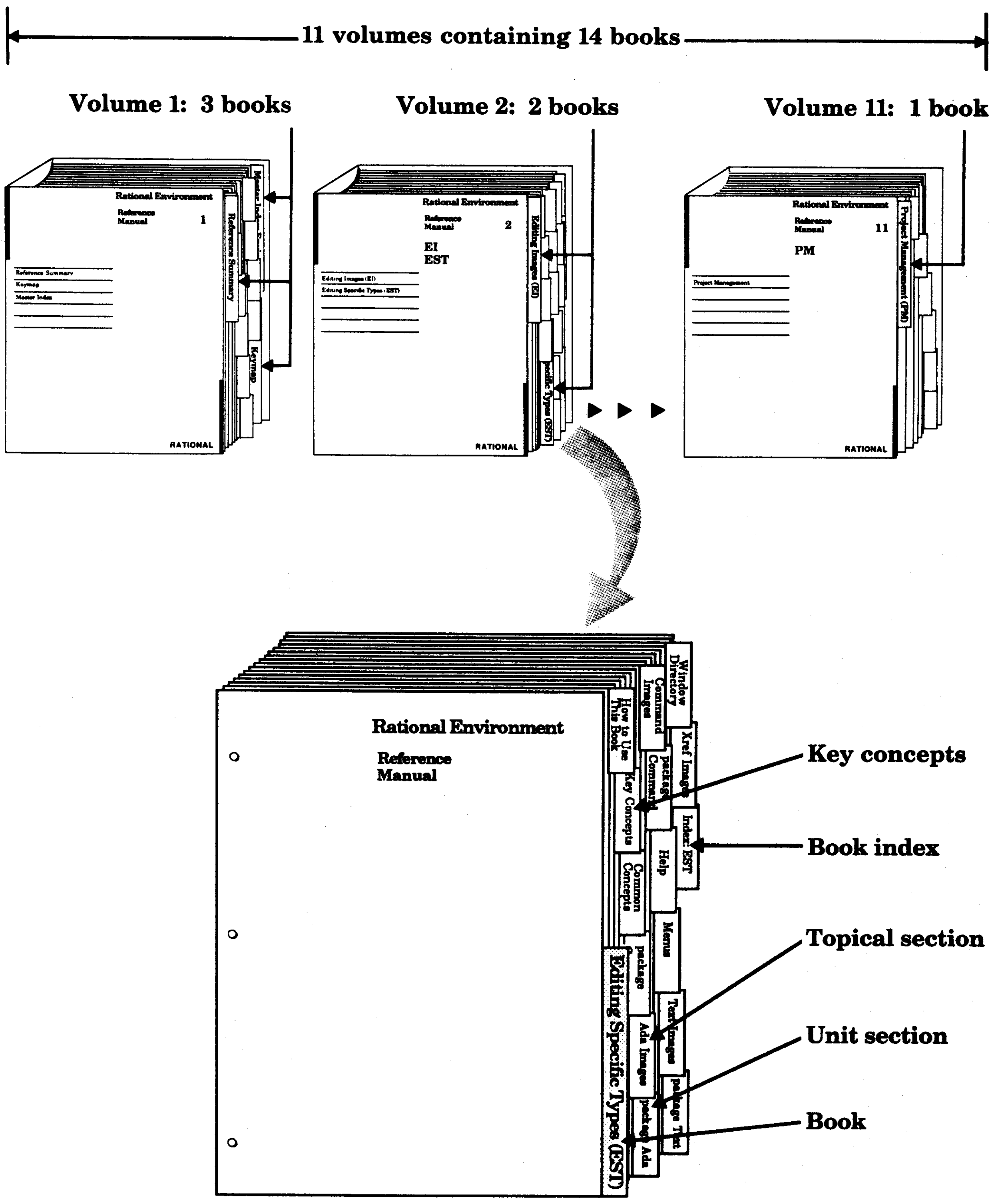
Organization of the Reference Manual

The *Rational Environment Reference Manual* (Reference Manual for brevity) includes the following volumes (see accompanying illustration):

1	Reference Summary
	Keymap
	Master Index
2	Editing Images (EI)
	Editing Specific Types (EST)
3	Debugging (DEB)
4	Session and Job Management (SJM)
5	Library Management (LM)
6	Text Input/Output (TIO)
7	Data and Device Input/Output (DIO)
8	String Tools (ST)
9	Programming Tools (PT)
10	System Management Utilities (SMU)
11	Project Management (PM)

Each *volume* of the Reference Manual contains one or more *books* separated by large colored tabs. Each book contains information on particular features or areas of application in the Environment. The abbreviation for the name of each book (for example, EI for Editing Images) appears on the binder cover and spine, and this abbreviation is used in page numbers and cross-references. The books grouped into one volume are not necessarily logically related.

Organization of the *Rational Environment Reference Manual*



A sample book

The Reference Manual provides reference information organized to efficiently answer specific questions about the Rational Environment. The *Rational Environment User's Guide* complements this manual, providing a user-oriented introduction to the facilities of the Environment. Products other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility) are documented in individual manuals, which are not part of the Reference Manual.

Volume 1

Volume 1, intended to be used as a quick reference to the resources provided by the Environment, contains the following books:

- **Reference Summary**: The Reference Summary contains the full Ada specification for each unit in the standard Environment. The unit specifications are organized by their pathnames. The World ! section provides a list of the units in the library system of the Environment and the manual/book in which they are documented.
- **Keymap**: The Rational Environment Keymap presents the standard Environment key bindings, organized by topic and by command name. The topical section includes both a quick reference for commonly used commands and a more detailed reference for key bindings.
- **Master Index**: The Master Index combines all of the index information for each of the books in the Reference Manual.

Volumes 2–11

Each book in Volumes 2–11 begins with a colored tab on which the name of the book appears. Each book typically contains the following sections:

- **Contents**: The table of contents provides a complete list of all the units in the book and their reference entries.
- **Key Concepts section**: Most of the books contain a section describing key concepts that pertain to all of the Environment facilities documented in that book. This section is located behind its own tab after the table of contents.
- **Unit sections**: Each of the commands, tools, and so on has a declaration within an Ada compilation unit (typically a package) in the Environment library system. For each unit, there is a section that contains reference entries for the declarations (for example, procedures, functions, and types) within that unit. Each section is preceded by a tab.

  The sections for units are alphabetized by the simple names of the units. For example, the section for package !Tools.String_Utilities is alphabetized under String_Utilities.

  For many units, introductory material and/or examples specific to the unit appear after the section tabs.

  Within the section for a given unit, the reference entries describing the unit's declarations are organized alphabetically after the section introduction. Appearing at the top of each page in a reference entry are the simple name of the given declaration and the fully qualified pathname of the enclosing unit.

- **Explanatory/topical sections**: Like the unit sections, explanatory/topical sections are preceded by tabs, and they are alphabetized with the unit sections. The topical sections, such as Help, located in Editing Specific Types (EST), discuss Environment facilities.
- **Index**: Preceded by a tab, the Index appears as the last section of each book. It contains entries for each unit or declaration, along with additional topical references. Each book index covers only the material documented in that particular book. The Master Index (in Volume 1) provides entries for the information documented in all the books within the Reference Manual.

  Italic page numbers indicate the page on which the primary reference entry for a declaration appears; nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Suggestions for Finding Information

The following suggestions may help you in finding various kinds of information in the documentation for Rational's products.

Learning about Environment Facilities

If you are a novice user starting to use the Environment, consult the *Rational Environment User's Guide*.

If you are familiar with the Environment but are interested in learning about the Environment's library-management commands, for example, you might start by scanning the specifications for these units in the Reference Summary to get an idea of the kinds of things these tools can do. You should also look at the Key Concepts for the particular book, which describes important concepts and gives examples.

It may also be useful to glance through the introductions provided for some of the units in the book. These introductions, located immediately after the tabs for the units, often contain helpful examples.

Finding Information on a Specific Item

If you know the name of the item and the book in which it is documented, consult either the table of contents or the index for that book. You can also turn through the pages of the book using the names and pathnames of the reference entries to locate the entry you want. Remember that the reference entries for a unit are organized alphabetically within the unit, and the units are organized alphabetically by simple name within the book.

If you know the simple name of the entry but do not know the book in which it is documented, look in the Master Index (in Volume 1) to find the book abbreviation and page number.

If you know the pathname of the entry but do not know the book in which it is documented, the World ! section of the Reference Summary (in Volume 1) provides a map of the units in the library system of the Environment and the books in which they are documented.

If you cannot find an item in the Master Index, the item either is not documented or is documented in the manuals for a product other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility). If you know the pathname, consult the World ! section of the Reference Summary to determine whether that item is documented and in which manual.

Using the Index

The index of each book contains entries for each unit and its declarations, organized alphabetically by simple name. When using the index to find a specific item, consult the italic page number for the primary reference for that item. Nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Viewing Specifications On-Line

If you know the pathname of a declaration and want to see its specification in a window of the Rational Environment, provide its pathname to the Common.Definition procedure—for example, `Definition ("!Commands.Library");`. If you know the simple name of the unit in which the declaration appears, in most cases you can use searchlist naming as a quick way of viewing the unit—for example, `Definition ("\Library");`.

Using On-Line Help

Most of the information contained in the reference entries for each unit is available through the on-line help facilities of the Environment. Press the [Help on Help] key or consult the *Rational Environment User's Guide* or the *Rational Environment Reference Manual*, EST, Help, for more information on using this on-line help facility.

Cross-Reference Conventions

The following conventions are used in cross-references to information:

- **Specific page/book**: For references to a specific place in a specific book, the book abbreviation is followed by the page number in the book (for example, LM-322). If the book abbreviation is omitted, the current book is implied (for example, the page numbers in the table of contents for a book do not include the book prefix).
- **Declaration in same unit**: References to the documentation for a declaration in the same unit are indicated by the simple name of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Library.Move procedure would be simply "procedure Move." Note that if there are nested packages in the unit, references to nested declarations use qualified pathnames.
- **Declaration in different unit, same book**: References to the documentation for a declaration in another unit are indicated by the qualified pathname of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Compilation.Delete procedure would be "procedure Compilation.Delete."

- **Declaration in different book**: References to the documentation for a declaration in another book are indicated by the addition of the abbreviation for that book. For example, within the reference entry for the Library.Copy procedure, a reference to the Editor.Region.Copy procedure in the Editing Images book would be "EI, procedure Editor.Region.Copy."

References to specific declarations in the library system of the Rational Environment (not the documentation for them) are typically indicated by fully qualified pathnames—for example, "procedure !Commands.Library.Copy." When the context is clear, however, a shorter name will be used. If the unit in which the declaration appears is undocumented, you may want to see its explanatory comments to understand what it does. To see these comments, either look at the unit's specification in the Reference Summary or view it on-line using the Rational Environment.

Feedback to Rational: Reader's Comments Form

Rational wants to make its documentation as useful and error-free as possible. Please provide us with feedback. The last page of each book contains a Reader's Comments form that you can use to send us comments or to report errors. You can also submit problem reports and make suggestions electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

package Bounded_String

Package Bounded_String provides a set of operations on objects of the Variable_String type. This type defines a dynamic string with a fixed maximum length. The length of the string is contained in the string itself. Note that the implementation of Variable_String type is not based on pointers or tasks; thus, objects of Variable_String type can be written into files.

The package contains several operations on the type that can raise the predefined Constraint_Error exception. Some operations can also raise the predefined Numeric_Error exception if invalid or uninitialized characters are passed to these operations. These two exceptions are the only exceptions or error conditions that operations in these packages can produce. The "Errors" section in the reference entry for each procedure or function specifies whether the operation can produce the exceptions and under what conditions.

procedure Append

```
procedure Append (Target : in out Variable_String;
                  Source :        Variable_String);

procedure Append (Target : in out Variable_String;
                  Source :        String);

procedure Append (Target : in out Variable_String;
                  Source :        Character);

procedure Append (Target : in out Variable_String;
                  Source :        Character;
                  Count  :        Natural);
```

Description

Appends the source character or characters to the target string.

This procedure appends characters to the end of an existing variable string. The length of the target variable string is increased by the number of the characters in the source variable string. The source can be another variable string, an Ada string, a single character, or a single character appended the number of times specified by the Count parameter.

Parameters

`Target : in out Variable_String;`

Specifies the object to which characters are to be appended. The length of the variable string is increased by the number of characters in the source.

`Source : Variable_String;`

Specifies a variable string to be appended to the target.

`Source : String;`

Specifies an Ada string to be appended to the target.

`Source : Character;`

Specifies a single character to be appended to the target.

`Count : Natural;`

Specifies the number of times the source character is to be appended to the target.

Errors

This procedure raises the Constraint_Error exception when the number of characters to be appended plus the current length of the target is greater than the maximum length of the target.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has length Ø
    Append (Command_Name, "Create ");
    -- Command_Name has length 7
    Append (Command_Name, 'x', 3);
    -- Command_Name has length 1Ø
```

function Char_At

```
function Char_At (Source : Variable_String;
                  At_Pos : Positive) return Character;
```

Description

Returns the character at the specified position in the source string.

Parameters

`Source : Variable_String;`

Specifies the variable string whose character is to be found.

`At_Pos : Positive;`

Specifies the position of the desired character in the source.

`return Character;`

Returns the character at the At_Pos position in the source variable string.

Errors

This function raises the Constraint_Error exception when the At_Pos position is greater than the current length of the source.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create yyy"
    ... Char_At (Command_Name, 3) -- returns 'e'
```

procedure Copy

```
procedure Copy (Target : in out Variable_String;
                Source :        Variable_String);

procedure Copy (Target : in out Variable_String;
                Source :        String);

procedure Copy (Target : in out Variable_String;
                Source :        Character);
```

Description

Copies the character(s) in the source variable string into the target variable string. The previous contents of the string are lost.

After execution of this procedure, the current length of the target variable string will be the length of the source.

Parameters

`Target :  in out Variable_String;`

Specifies the variable string in which to place the copied characters. The previous contents of the string are lost. After execution of this procedure, the current length of the target variable string will be the length of the source.

`Source :  Variable_String;`

`Source :  String;`

`Source :  Character;`

Specifies the source from which to copy characters into the target.

Errors

This procedure raises the Constraint_Error exception when the current length of the source string is greater than the maximum length of the target string.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

The procedure:

```
    Copy (Command_Name, "Create");
```

copies the six characters of the string into the Command_Name string; thus, the new length of Command_Name is 6. The previous contents of Command_Name are lost.

---

procedure Delete

```
procedure Delete (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Count  :        Natural          := 1);
```

Description

Deletes the character(s) from the specified position in the target variable string.

This procedure deletes characters from the At_Pos position in the variable string. The length of the target is decreased by the value of the Count parameter.

Parameters

`Target :  in out Variable_String;`

Specifies the variable string from which characters are to be deleted.

`At_Pos :  Positive;`

Specifies the position in the target from which characters are to be deleted.

`Count :  Natural := 1;`

Specifies the number of characters to be deleted. The default is 1.

Errors

This procedure raises the Constraint_Error exception when the value of the Count parameter is greater than the current length of the target minus the position in the target.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create vvv xxx"
    Delete (Command_Name, 8, 4);
    -- Command_Name has contents "Create xxx"
```

function Extract

```
function Extract (Source    : Variable_String;
                  Start_Pos : Positive;
                  End_Pos   : Natural) return String;
```

Description

Extracts a string from the starting position to the ending position from the source variable string.

This function returns the portion of the variable string that begins with the character at the Start_Pos position and ends with the character at the End_Pos position. The returned string has an index of (Start_Pos .. End_Pos). When the value of End_Pos is less than the value of Start_Pos, the returned string is null.

Parameters

`Source : Variable_String;`

Specifies the variable string from which a string is to be extracted.

`Start_Pos : Positive;`

Specifies the beginning position in the source from which a string is to be extracted.

`End_Pos : Natural;`

Specifies the ending position in the source from which a string is to be extracted.

`return String;`

Returns the extracted string with an index of (Start_Pos .. End_Pos).

Errors

This function raises the Constraint_Error exception when either the Start_Pos or the End_Pos value is greater than the current length of the source.

Example

```
-- Command_Name has contents "Create yyy"
... Extract (Command_Name, 8, 10) -- returns "yyy"
```

procedure Free

```
procedure Free (V : in out Variable_String);
```

Description

Frees the specified variable string, setting its length to 0.

This procedure frees the specified object by setting its length to 0. The previous contents of the object are lost.

Parameters

`V : in out Variable_String;`

Specifies the object to be freed. Its previous contents are lost.

Errors

This procedure does not raise any exceptions.

function Image

```
function Image (V : Variable_String) return String;
```

Description

Converts the variable string into an Ada predefined string.

This function converts an object of the Variable_String type to the String type. If the input has a length of 0, then the result is the null string. The result has the same length as the current length of the input. The result has an index of 1 to the current length of the input.

Parameters

`V : Variable_String;`
Specifies the object to be converted to the String type.

`return String;`
Returns the contents of the variable string in an object of the String type.

Errors

This function does not raise any exceptions.

procedure Insert

```
procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        Variable_String);

procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        String);

procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        Character);

procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        Character;
                  Count  :        Natural);
```

Description

Inserts the specified character(s) from the source variable string into the specified position in the target variable string.

This procedure inserts characters into the target, increasing the length of the target by the number of characters in the source. The characters are inserted between the character at the At_Pos position and the character immediately before that position. The source can be another variable string, an Ada string, a single character, or a single character inserted the number of times specified in the Count parameter.

Parameters

```
Target :  in out Variable_String;
```

Specifies the variable string in which characters from the source are to be inserted.

```
At_Pos :  Positive;
```

Specifies the position within the target at which characters are to be inserted.

```
Source :  Variable_String;
```

Specifies the variable string to be inserted into the target.

```
Source :  String;
```

Specifies the string to be inserted into the target.

```
Source : Character;
```
Specifies the character to be inserted into the target.

```
Count : Natural;
```
Specifies the number of times the character is to be inserted into the target.

Errors

This procedure raises the Constraint_Error exception when the number of characters to be inserted plus the current length of the target is greater than the maximum length of the target.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create xxx"
    Insert (Command_Name, 8, "vvv ");
    -- Command_Name has contents "Create vvv xxx"
```

---

function Length

```
function Length (Source : Variable_String) return Natural;
```

Description

Returns the current length of the source variable string.

Parameters

```
Source :  Variable_String;
```
Specifies the variable string whose length is to be found.

```
return Natural;
```
Returns the length of the source.

Errors

This function does not raise any exceptions.

Example

```
    -- Command_Name has contents "Create yyy"
    ... Length (Command_Name) -- returns 10
```

function Max_Length

```
function Max_Length (Source : Variable_String) return Natural;
```

Description

Returns the maximum length of the source variable string.

Parameters

```
Source : Variable_String;
```

Specifies the variable string whose maximum length is to be found.

```
return Natural;
```

Returns the maximum length of the source.

Errors

This function does not raise any exceptions.

procedure Move

```
procedure Move (Target : in out Variable_String;
                Source : in out Variable_String);
```

Description

Moves the contents of the source variable string into the target variable string, destroying the current contents of the source.

This procedure is equivalent to copying the source into the target and then calling the Free procedure on the target.

Parameters

```
Target :  in out Variable_String;
```

Specifies the variable string into which characters from the source are to be moved. The previous contents are destroyed. After the operation, the length of the variable string is the same as the length of the source before the operation.

```
Source :  in out Variable_String;
```

Specifies the variable string from which characters are to be moved to the target. Moving the contents of the source to the target destroys them in the source. After the operation, the string has a length of 0.

Errors

The copy portion of the operation can raise the Constraint_Error exception if the current length of the source string is greater than the maximum length of the target string.

procedure Replace

```
procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        Character);

procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        Character;
                   Count  :        Natural);

procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        String);

procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        Variable_String);
```

Description

Replaces character(s) in the target variable string with character(s) from the source variable string.

The number of characters replaced is the number of characters in the source. The length of the target remains the same. The source can be another variable string, an Ada string, a single character, or a single character the repeated number of times specified by the Count parameter.

Parameters

`Target : in out Variable_String;`

Specifies the variable string in which characters are to be replaced from the source.

`At_Pos : Positive;`

Specifies the position within the target at which characters are to be replaced.

`Source : Character;`

Specifies the character to be replaced in the target.

`Source : String;`

Specifies the Ada string to be replaced in the target.

```
Source :  Variable_String;
```

Specifies the variable string to be replaced in the target.

```
Count :  Natural;
```

Specifies the number of times the character in the target is to be replaced.

Errors

This procedure raises the Constraint_Error exception when the number of characters to be replaced plus the position is greater than the current length of the target.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create xxx"
    Replace (Command_Name, 8, "yyy");
    -- Command_Name has contents "Create yyy"
```

procedure Set_Length

```
procedure Set_Length (Target     : in out Variable_String;
                      New_Length :        Natural;
                      Fill_With  :        Character        := ' ');
```

Description

Truncates or fills the variable string to the specified length.

This procedure sets the length of the target variable string to the value specified in the New_Length parameter. When the new length is less than the old length, the variable string is truncated. When the new length is the same as the old length, the procedure has no effect. When the new length is greater than the old length, the variable string is extended with the new character positions filled with the Fill_With parameter.

Parameters

`Target : in out Variable_String;`

Specifies the variable string whose length is to be changed.

`New_Length : Natural;`

Specifies the new length of the target.

`Fill_With : Character := ' ';`

Specifies the fill character. This character is significant only if the target is made larger. The default is a space character.

Errors

This procedure raises the Constraint_Error exception when the new length is greater than the maximum length of the target.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
-- Command_Name has contents "Create yyy"
Set_Length (Command_Name, 12);
-- Command_Name has contents "Create yyy  "
```

subtype String_Length

```
subtype String_Length is Natural;
```

Description

Defines the allowed values of the length of the variable string.

The length of the variable string can be between 0 and Integer'Last, inclusive.

function Value

```
function Value (S : String) return Variable_String;

function Value (S          : String;
                Max_Length : Natural) return Variable_String;
```

Description

Converts the string into a variable string.

This function converts an object of the String type to the Variable_String type. The current length of the result is the same length as the length of the input string. The result has a maximum length the same as the length of the string or as specified in the Max_Length parameter.

Parameters

```
S : String;
```

Specifies the object to be converted to the Variable_String type.

```
Max_Length : Natural;
```

Specifies the maximum length of the Variable_String returned by this function. If the specified length is less than the current length of the input string, the Constraint_Error exception is raised.

```
return Variable_String;
```

Returns the contents of the input string in an object of Variable_String type.

Errors

The second form of this function raises the Constraint_Error exception when the specified maximum length is less than the current length of the input string.

type Variable_String

```
type Variable_String (Maximum_Length : String_Length) is private;
```

Description

Defines the abstract type.

The string's maximum length is determined by the discriminant. Upon elaboration, an object of this type has a current length of zero characters.

Implementation of the Variable_String type is not based on pointers or tasks. This means that objects of the Variable_String type can be written into files.

Example

```
    Command_Name : Variable_String(20);
```

Defines an object, Command_Name, of this type with a current length of zero characters and a maximum length of 20 characters.

end Bounded_String;

RATIONAL

generic package String_Map_Generic

Generic package String_Map_Generic provides a means of mapping string values to values of another type. This translation from strings (called the *domain type*) to another type (called the *range type*) is a one-to-one mapping.

Some operations can perform optional checks for domain values that are already defined. For these operations a parameter selects the action that should be taken when a supplied domain value is already defined. The operation can complete or it can be aborted, raising the Multiply_Defined exception (in this package). Also, if any operation is passed illegal input, the Constraint_Error exception is raised.

The implementation of this generic package uses a hash table. The size of the hash table (the number of buckets) is one of the formal parameters of the generic. The hash function is built into the generic for dividing the string values of the domain into the buckets.

The formal parameters to the generic are:

```
generic
    Size : in Integer;
    type Range_Type is private;
    Ignore_Case : Boolean := True;
package String_Map_Generic is
    ...
end String_Map_Generic;
```

function Cardinality

```
function Cardinality (The_Map : Map) return Natural;
```

Description

Returns the number of mappings defined in The_Map parameter.

Parameters

`The_Map : Map;`

Specifies the map to be checked.

`return Natural;`

Returns the number of mappings defined in the map.

Errors

This function raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

procedure Copy

```
procedure Copy (Target : in out Map;
                Source :        Map);
```

Description

Copies the contents of the source map into the target map.

This procedure first deletes all entries from the target and then adds an entry in the target for each entry in the source.

Parameters

`Target : in out Map;`
Specifies the map into which entries are to be copied.

`Source : Map;`
Specifies the map from which entries are to be copied.

Errors

This procedure raises the Constraint_Error exception when the value of either map is an uninitialized map.

procedure Define

```
procedure Define (The_Map        : in out Map;
                  D              :        String;
                  R              :        Range_Type;
                  Trap_Multiples :        Boolean     := False);
```

Description

Defines a correspondence between a string value and a range value.

This procedure creates entries in the map. The procedure traps multiple definitions of string values when the Trap_Multiples parameter is true. When the parameter is false, the string value is redefined to the new range value.

Parameters

`The_Map :  in out Map;`

Specifies the map to which the entry is to be added.

`D : String;`

Specifies the string value to be added to the map.

`R : Range_Type;`

Specifies the range value that corresponds to the string value.

`Trap_Multiples  :  Boolean := False;`

Specifies whether to trap an attempt to create a multiple definition of a string value. When the parameter is true and a string value that already exists in the map is given, the Multiply_Defined exception is raised. The default is not to trap multiple definitions.

Errors

This procedure raises the Multiply_Defined exception (in this package) when the value of the Trap_Multiples parameter is true and a string value that already exists in the map is given.

The procedure also raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Determines whether the iteration over the map is complete.

Parameters

```
Iter :  Iterator;
```

Specifies the iteration to be checked.

```
return Boolean;
```

Returns the value true when the iteration is complete; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
    Init (Command_Iterator, Command_Map);
    while not Done (Command_Iterator) loop
        ...
        ... Value (Command_Iterator)
        ...
        Next (Command_Iterator);
    end loop;
```

function Eval

```
function Eval (The_Map : Map;
               D       : String) return Range_Type;
```

Description

Finds the corresponding value of the range given a map and a string value.

This function returns a value of the range that corresponds to the string value in the map. When the given value of the string does not exist, the Undefined exception is raised.

Parameters

```
The_Map :  Map;
```

Specifies the map in which the string value is to be found.

```
D : String;
```

Specifies the string value to be found in the map.

```
return Range_Type;
```

Returns the corresponding value of the range for the given string value.

Errors

This function raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

The function also raises the Undefined exception (in this package) when the string value does not exist in the map.

procedure Find

```
procedure Find (The_Map :        Map;
                D       :        String;
                R       : in out Range_Type;
                Success :    out Boolean);
```

Description

Finds the range value corresponding to the specified domain value (D) in the specified map.

This procedure finds the value of the range that corresponds to the string value in the map. When the given value of the string exists in the map, the Range parameter is updated with the range value, and the Success parameter is returned true. When the given string value does not exist, the Range parameter is not changed, and the Success parameter is returned false.

Parameters

`The_Map : Map;`

Specifies the map in which the domain value is to be found.

`D : String;`

Specifies the string value to be found in the map.

`R : in out Range_Type;`

Returns the value of the range when the string value is found. The parameter is not changed when the string value is not found.

`Success : out Boolean;`

Returns the value true when the string value exists in the map; otherwise, the procedure returns false.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

generic formal object Ignore_Case

```
Ignore_Case : Boolean := True;
```

Description

Specifies whether comparisons on domain string values are case-sensitive.

By default, all domain value comparisons are not case-sensitive.

procedure Init

```
procedure Init (Iter     : out Iterator;
                The_Map  :     Map);
```

Description

Initializes the iterator to iterate through all the entries in the specified map.

This procedure initializes the iterator for the specified map. When one or more entries exist in the map, the Value function returns the first entry in the map using this value of the iterator. When no entries exist in the map, the Done function returns the value true using this value of the iterator.

Parameters

`Iter : out Iterator;`

Returns the iterator that can be used to iterate through all the entries in the map.

`The_Map : Map;`

Specifies the map for which the iterator is desired.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

Example

This example demonstrates use of the iteration capability:

```
Init (Command_Iterator, Command_Map);
while not Done (Command_Iterator) loop
    ...
    ... Value (Command_Iterator)
    ...
    Next (Command_Iterator);
end loop;
```

References

function Done

function Value

procedure Initialize

```
procedure Initialize (The_Map : out Map);
```

Description

Creates an initialized and empty map.

This procedure creates a map; the map does not have to be created by the Nil function. The map contains no entries.

Parameters

```
The_Map :  out Map;
```

Specifies the value of the map to be created.

References

function Is_Nil

function Nil

function Is_Empty

```
function Is_Empty (The_Map : Map) return Boolean;
```

Description

Checks whether the specified map is empty.

This function checks whether the map contains no entries. When a map is initialized or after the Make_Empty procedure is executed on the map, this condition is true.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Boolean;
```

Returns the value true when no entries exist in the map; otherwise, the function returns false.

Errors

This function raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

References

procedure Make_Empty

function Is_Nil

```
function Is_Nil (The_Map : Map) return Boolean;
```

Description

Checks whether the map is initialized.

This function checks the specified map to determine whether it has been initialized. The map must be initialized before being used with other procedures and functions in this package.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Boolean;
```

Returns the value true when the map is uninitialized; otherwise, the function returns false.

References

procedure Initialize

function Nil

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all entries in a map.

Objects of the Iterator type contain all of the information necessary to step over all of the entries in a map. The type is used with the Init and Next procedures and the Value and Done functions.

procedure Make_Empty

```
procedure Make_Empty (The_Map : in out Map);
```

Description

Clears the map of all entries.

Parameters

```
The_Map :  in out Map;
```

Specifies the map to be cleared.

Errors

This procedure raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

type Map

```
type Map is private;
```

Description

Defines the representation of the map.

Several important properties of the Map type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the map are not copied but a new alias (a new name) for the map is created. Equality for this type has the property that the values of the maps are not compared but the names are compared. In other words, the operation of equality checks to determine whether any two map values designate the same map.

exception Multiply_Defined

```
Multiply_Defined : exception;
```

Description

Defines the exception raised by the Define procedure when the Domain parameter is already defined in the specified map and the value of the Trap_Multiples parameter is true.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Steps the iterator to point to the next entry in the map.

This procedure changes the iterator to point to the next entry in the map. When the iterator steps past the last entry, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
    Init (Command_Iterator, Command_Map);
    while not Done (Command_Iterator) loop
        ...
        ... Value (Command_Iterator)
        ...
        Next (Command_Iterator);
    end loop;
```

References

function Done

procedure Init

function Value

function Nil

```
function Nil return Map;
```

Description

Returns an uninitialized map.

This function creates a new map. The map is uninitialized, and it must be initialized before it can be used.

Parameters

```
return Map;
```

Returns the map.

References

procedure Initialize

function Is_Nil

generic formal type Range_Type

```
type Range_Type is private;
```

Description

Defines the type for the range of the map.

The type must be a pure value type, and it must be constrained. The implicit operations of assignment and equality must work with values of this type.

generic formal object Size

```
Size : Integer;
```

Description

Defines the size of the hash table in the map.

The value provided for this generic parameter depends on the number of expected or allowed values of the string domain. Preferred values are prime numbers.

procedure Undefine

```
procedure Undefine (The_Map : in out Map;
                    D       :        String);
```

Description

Removes a map entry or entries for the string value.

This procedure removes all entries for the specified string value from the map. When the string value does not exist in the map, the procedure raises the Undefined exception.

Parameters

`The_Map :  in out Map;`

Specifies the map from which the entry is to be removed.

`D : String;`

Specifies the string value to be removed.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

The procedure raises the Undefined exception (in this package) when the string value does not exist in the map.

exception Undefined

```
Undefined : exception;
```

Description

Defines the exception raised by the Eval function or the Undefine procedure when the Domain parameter does not exist in the specified map.

function Value

```
function Value (Iter : Iterator) return String;
```

Description

Returns the string value of the current entry pointed to by the iterator.

Parameters

```
Iter :  Iterator;
```

Specifies the iteration from which the string value is to be returned.

```
return String;
```

Returns the string value.

Example

This example demonstrates use of the iteration capability:

```
    Init (Command_Iterator, Command_Map);
    while not Done (Command_Iterator) loop
        ...
        ... Value (Command_Iterator)
        ...
        Next (Command_Iterator);
    end loop;
```

end String_Map_Generic;

package String_Table

Package String_Table provides a set of operations on tables of unique strings. The string table can be of any size, although the size of the hash table for the strings is specified when the string table is created. The operations of this package assure that the table does not contain duplicate strings.

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to these operations. If other illegal parameters are passed to any of the operations in this package, the Constraint_Error exception is raised. If any of the operations that add entries to the table are attempted when the table is full, the Table_Full exception (in this package) is raised.

function Allocate

```
function Allocate (Source   : String;
                   In_Table : Table) return Item;
```

Description

Creates an item for the specified string in the specified table.

This function returns an item that represents the specified string. The string is not inserted into the table.

Parameters

`Source :  String;`

Specifies the string for which an item is desired.

`In_Table :  Table;`

Specifies the table in which the item is desired.

`return Item;`

Returns the item.

function Char_At

```
function Char_At (Source : Item;
                  At_Pos : Natural) return Character;
```

Description

Returns the specified character from the unique string.

This function provides access to individual characters in the corresponding string of the item.

Parameters

`Source :  Item;`

Specifies the item from which the character from the corresponding string is desired.

`At_Pos :  Natural;`

Specifies the position in the string at which the character is desired.

`return Character;`

Returns the character.

Errors

This function raises the Constraint_Error exception when the position specified is outside the bounds of the corresponding string or when the item specified is nil.

The function raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Checks whether the iteration over the table is complete.

This function checks whether the iterator has stepped through all of the entries in the table.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator to be checked.

```
return Boolean;
```

Returns the value true when the iterator has stepped through all of the entries in the table; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
    Init (Command_Iterator, Command_Map);
    while not Done (Command_Iterator) loop
        ...
        ... Value (Command_Iterator)
        ...
        Next (Command_Iterator)
    end loop;
```

function Equal

```
function Equal (L : Item;
                R : Item) return Boolean;
```

Description

Determines whether the two unique strings are equal.

Parameters

`L : Item;`

`R : Item;`
Specifies a string to check.

`return Boolean;`
Returns true if the input strings are equal; otherwise, the function returns false.

function Find

```
function Find (Source      : String;
               In_Table    : Table;
               Ignore_Case : Boolean := True) return Item;
```

Description

Searches the table for the unique string.

This function searches the table for the string. When the string is in the table, its corresponding item is returned. When the string is not in the table, the nil item is returned.

Parameters

`Source : String;`

Specifies the string to be found in the table.

`In_Table : Table;`

Specifies the table in which to search for the string.

`Ignore_Case : Boolean := True;`

Specifies whether to ignore the case of the string. The default, true, is to ignore the case.

`return Item;`

Returns the corresponding item of the string when it exists in the table; otherwise, the function returns the nil item.

function Image

```
function Image (Source : Item) return String;
```

Description

Returns the corresponding string of the item.

Parameters

```
Source :  Item;
```

Specifies the item for which the corresponding string is desired.

```
return String;
```

Returns the string.

procedure Init

```
procedure Init (Iter      : out Iterator;
                The_Table :     Table);
```

Description

Initializes the iterator to iterate through all the items in the specified table.

When one or more entries exist in the table, the Value function returns the first entry in the table using this value of the iterator. When no entries exist in the table, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter : out Iterator;
```

Returns the iterator.

```
The_Table : Table;
```

Specifies the table for which the iterator is desired.

Example

This example demonstrates use of the iteration capability:

```
    Init (Command_Iterator, Command_Map);
    while not Done (Command_Iterator) loop
        ...
        ... Value (Command_Iterator)
        ...
        Next (Command_Iterator);
    end loop;
```

References

function Done

procedure Next

function Value

function Is_Nil

```
function Is_Nil (Source : Item) return Boolean;
```

Description

Determines whether the specified item is equal to nil.

Parameters

```
Source :  Item;
```

Specifies the item to be checked for nil.

```
return Boolean;
```

Returns the value true when the source is equal to nil; otherwise, the function returns false.

type Item

```
type Item is private;
```

Description

Defines the type used to represent strings in the table.

The Item type provides a handle for manipulating strings. Typically, strings are added to the table and item values are returned. The strings can then be manipulated by passing these values to operations in this package.

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all entries in a table.

Objects of this type contain all of the information necessary to step over all of the entries in a table. The type is used with the Init and Next procedures and the Value and Done functions.

function Length

```
function Length (Source : Item) return Natural;
```

Description

Returns the length of the corresponding string of the item.

Parameters

```
Source : Item;
```

Specifies the item for which the length of the corresponding string is desired.

```
return Natural;
```

Returns the length of the corresponding string.

Errors

This function raises the Constraint_Error exception when the item specified is nil.

function New_Table

```
function New_Table (Minimum_Table_Size : Natural := 127) return Table;
```

Description

Creates an empty table of the specified size.

This function creates a string table of the specified minimum size. The table initially has no strings in it. The parameter specifies the hash table size for the new table.

Parameters

```
Minimum_Table_Size : Natural := 127;
```

Specifies the hash table size for the new table. The value should be a prime number. The default is 127.

```
return Table;
```

Returns the table.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Steps the iterator to point to the next entry in the table.

This procedure changes the iterator to point to the next entry in the table. Although the table is unordered, the iterator steps through all entries one by one. When the iterator steps past the last entry, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
    Init (Command_Iterator, Command_Map);
    while not Done (Command_Iterator) loop
        ...
        ... Value (Command_Iterator)
        ...
        Next (Command_Iterator)
    end loop;
```

References

function Done

procedure Init

function Value

function Nil

```
function Nil return Item;
```

Description

Returns a nil item.

This function returns an item that does not represent any possible string. When the Find function determines that the specified string is not in the table, this value of the Item type is returned.

Parameters

```
return Item;
```

Returns the nil item.

References

function Find

function Is_Nil

type Table

```
type Table is private;
```

Description

Defines the representation of the table.

Several important properties of the Table type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the table are not copied but a new alias (a new name) for the table is created. Equality for this type has the property that the values of the tables are not compared but the names are compared. In other words, the operation of equality checks to determine whether any two table values designate the same table.

exception Table_Full

```
Table_Full : exception;
```

Description

Defines an error condition that results when an operation attempting to add an entry to a table finds the table full.

function Unique

```
function Unique (Source      : String;
                 In_Table    : Table;
                 Ignore_Case : Boolean := True) return Item;
```

Description

Inserts, if necessary, the specified unique string in the specified table and returns the corresponding item.

This function checks whether the source string is in the table. When the source string is in the table, the corresponding item is returned. When the string is not in the table, the string is added to the table and the corresponding item of the inserted string is returned.

Parameters

`Source : String;`

Specifies the string to be checked for uniqueness and to be added to the table, if necessary.

`In_Table : Table;`

Specifies the table in which the source string is to be found.

`Ignore_Case : Boolean := True;`

Specifies whether to ignore the case of the string. The default, true, is to ignore the case. The strings are stored in uppercase when the parameter is true.

`return Item;`

Returns the corresponding item of the string.

package String_Utilities

Package String_Utilities provides a series of utilities that operate on the Ada predefined String type. These utilities include:

- A hash function
- Case conversions
- Numeric conversions
- Cleaning and stripping utilities
- Searches
- Comparisons

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to these operations.

function Capitalize

```
function Capitalize (S : String) return String;
```

Description

Returns the source string with the first letter of each word capitalized.

Parameters

```
S : String;
```

Specifies the string to be capitalized.

```
return String;
```

Returns the input string with the first letter of each word capitalized. Each letter after any character that is not a letter or a digit will be changed to uppercase.

References

function Upper_Case

procedure Capitalize

```
procedure Capitalize (S : in out String);
```

Description

Capitalizes the first letter of each word in the input string.

Parameters

```
S : in out String;
```

Returns the input string with the first letter of each word capitalized. Each word in the returned string will be capitalized—that is, each letter after any character that is not a letter or a digit will be changed to uppercase.

References

procedure Upper_Case

function Equal

```
function Equal (Str1        : String;
                Str2        : String;
                Ignore_Case : Boolean := True) return Boolean;
```

Description

Compares two strings for equality.

This function compares two strings in either a case-sensitive or case-insensitive way. When the two strings are equal, the function returns true. This function is the same as the Ada predefined equality operation on strings, except that it can be case-insensitive.

Parameters

`Str1 :  String;`

Specifies the first string in the comparison.

`Str2 :  String;`

Specifies the second string in the comparison.

`Ignore_Case :  Boolean := True;`

Specifies whether to do a case-sensitive or case-insensitive comparison. The default, true, is to do a case-insensitive comparison.

`return Boolean;`

Returns the value true when the two strings are equal; otherwise, the function returns false.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

function Greater_Than

```
function Greater_Than (Str1        : String;
                       Str2        : String;
                       Ignore_Case : Boolean := True) return Boolean;
```

Description

Checks whether one string is greater than another string.

This function does an ASCII comparison of two strings. The strings are compared from left to right, one character at a time. When a character from the first string has an ASCII character code that is greater than the corresponding character from the second string, the function returns the value true. Otherwise, the function returns the value false.

Parameters

`Str1 :  String;`

Specifies the first string in the comparison.

`Str2 :  String;`

Specifies the second string in the comparison.

`Ignore_Case :  Boolean := True;`

Specifies whether to do a case-sensitive or case-insensitive comparison. The default, true, is to do a case-insensitive comparison.

`return Boolean;`

Returns the value true when the first string is greater than the second string; otherwise, the function returns false.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

function Hash_String

```
function Hash_String (S : String) return Integer;
```

Description

Hashes the specified string into an integer.

Parameters

```
S : String;
```

Specifies the string to be hashed.

```
return Integer;
```

Returns an integer.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

function Less_Than

```
function Less_Than (Str1        : String;
                    Str2        : String;
                    Ignore_Case : Boolean := True) return Boolean;
```

Description

Checks whether one string is less than another string.

This function does an ASCII comparison of two strings. The strings are compared from left to right, one character at a time. When a character from the first string has an ASCII character code that is less than the corresponding character from the second string, the function returns the value true. Otherwise, the function returns the value false.

Parameters

`Str1 :  String;`

Specifies the first string in the comparison.

`Str2 :  String;`

Specifies the second string in the comparison.

`Ignore_Case :  Boolean := True;`

Specifies whether to do a case-sensitive or case-insensitive comparison. The default, true, is to do a case-insensitive comparison.

`return Boolean;`

Returns the value true when the first string is less than the second string; otherwise, the function returns false.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

function Locate

```
function Locate (Fragment    : String;
                 Within      : String;
                 Ignore_Case : Boolean := True) return Natural;

function Locate (Fragment    : Character;
                 Within      : String;
                 Ignore_Case : Boolean    := True) return Natural;
```

Description

Finds the character or string fragment in the string from left to right.

This function searches the specified string for the character or string fragment. The search begins with the first character in the string and proceeds from left to right. When the character or string fragment is found, the function returns the index (not the location) in the string of the character or the first character of the string fragment. When the character or string fragment is not found, the function returns the value of 0.

Parameters

`Fragment : String;`

Specifies the string fragment to be found in the string.

`Fragment : Character;`

Specifies the character to be found in the string.

`Within : String;`

Specifies the string in which to search.

`Ignore_Case : Boolean := True;`

Specifies whether to ignore the case of all characters compared in the search. The default, true, is to do a case-insensitive search.

`return Natural;`

Returns the index (not the position) of the character or string fragment, if found; otherwise, the function returns 0.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
Locate ("ab", "abcabc")
-- returns the value 1
```

Note that if the string searched had bounds different from `1 .. 6`, the value returned would be the index of the lower bound of the string searched rather than the value 1 shown in this example.

function Lower_Case

```
function Lower_Case (C : Character) return Character;

function Lower_Case (S : String) return String;
```

Description

Returns the specified character or string in all lowercase characters.

Parameters

`C : Character;`
Specifies the character to be returned in a lowercase character.

`S : String;`
Specifies the string to be returned in lowercase characters.

`return Character;`
Returns the lowercase character.

`return String;`
Returns the lowercase string.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

procedure Lower_Case

```
procedure Lower_Case (C : in out Character);

procedure Lower_Case (S : in out String);
```

Description

Changes the specified character or string to all lowercase characters.

Parameters

`C : in out Character;`
Specifies the character to be changed to a lowercase character.

`S : in out String;`
Specifies the string to be changed to lowercase characters.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

function Number_To_String

```
function Number_To_String (Value   : Integer;
                           Base    : Natural   := 10;
                           Width   : Natural   := 0;
                           Leading : Character := ' ') return String;

function Number_To_String (Value   : Long_Integer;
                           Base    : Natural       := 10;
                           Width   : Natural       := 0;
                           Leading : Character     := ' ') return String;
```

Description

Converts the specified value to a string of the specified base padded to the specified width.

This function converts a value to a string. The string is created in the specified base. The resulting string is padded to the specified width with the specified leading character, if necessary. Even if the string resulting from the conversion without padding is more than the specified width, the entire string will be returned.

Parameters

`Value : Integer;`

`Value : Long_Integer;`

Specifies the value to be converted.

`Base : Natural := 10;`

Specifies the base, or radix, of the string to be created. Allowed values are 2 through 16, inclusive. The default is decimal.

`Width : Natural := 0;`

Specifies the minimum number of characters to be produced by the conversion through padding of the beginning of the string with the specified leading character. The default is not to pad the string that results from the conversion.

`Leading : Character := ' ';`

Specifies a character to be inserted on the left of the string to complete the string to the specified width. The default is a space character.

```
return String;
```

Returns the string that represents the number.

function Reverse_Locate

```
function Reverse_Locate (Fragment    : String;
                         Within      : String;
                         Ignore_Case : Boolean := True) return Natural;

function Reverse_Locate (Fragment    : Character;
                         Within      : String;
                         Ignore_Case : Boolean     := True) return Natural;
```

Description

Finds the character or string fragment in the string from right to left.

This function searches the specified string for the character or string fragment. The search begins with the last character in the string and proceeds from right to left. When the character or string fragment is found, the function returns the index (not the location) in the string of the character or the last character of the string fragment. When the character or string fragment is not found, the function returns the value of 0.

Parameters

`Fragment : String;`

Specifies the string fragment to be found in the string.

`Fragment : Character;`

Specifies the character to be found in the string.

`Within : String;`

Specifies the string in which to search.

`Ignore_Case : Boolean := True;`

Specifies whether to ignore the case of all characters compared in the search. The default, true, is to do a case-insensitive search.

`return Natural;`

Returns the index (not the position) of the character or string fragment, if found; otherwise, the function returns 0.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    Reverse_Locate ("ab", "abcabc")
    -- returns the value 5
```

Note that if the string searched had bounds different from `1 .. 6`, the value returned would be the index of the second `b` character in the string searched, not necessarily `5` as in this example.

procedure String_To_Number

```
procedure String_To_Number (Source :     String;
                            Target : out Integer;
                            Worked : out Boolean;
                            Base   :     Natural  := 10);

procedure String_To_Number (Source :     String;
                            Target : out Long_Integer;
                            Worked : out Boolean;
                            Base   :     Natural       := 10);
```

Description

Converts the specified string with the specified base to an integer value.

This procedure converts a string to a value. The string is evaluated in the specified base. When the conversion is completed correctly, the Worked parameter is returned true and the resulting integer is returned in the Target parameter.

Parameters

`Source : String;`

Specifies the string to be converted.

`Target : out Integer;`

`Target : out Long_Integer;`

Returns the value of the string.

`Worked : out Boolean;`

Returns the value true when the conversion is completed correctly; otherwise, the procedure returns false. Conversion will not succeed if the string is not a number or if the number is out of range. Note that the number cannot be a based literal.

`Base : Natural := 10;`

Specifies the base, or radix, of the string. Allowed values are 2 through 16, inclusive. The default is decimal.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

---

function Strip

```
function Strip (From   : String;
                Filler : Character := ' ') return String;
```

Description

Removes zero or more characters from both the beginning and the end of the specified string.

This function strips leading and trailing characters from a string. Zero or more of the specified characters are stripped from the beginning and from the end of the string. When the first and last characters of the string are not the specified characters, the string is returned unchanged.

Parameters

`From : String;`

Specifies the string from which characters are to be stripped.

`Filler : Character := ' ';`

Specifies the character to be stripped from the string. The default is to strip leading and trailing spaces.

`return String;`

Returns the stripped string.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    Strip ("  this is a test ... ")
    -- returns "this is a test ..."
```

function Strip_Leading

```
function Strip_Leading (From   : String;
                        Filler : Character := ' ') return String;
```

Description

Removes zero or more characters from the beginning of the specified string.

This function strips leading characters from a string. Zero or more of the specified characters are stripped from the beginning of the string. When the first character of the string is not the specified character, the string is returned unchanged.

Parameters

`From :  String;`
Specifies the string from which characters are to be stripped.

`Filler :  Character := ' ';`
Specifies the character to be stripped from the string. The default is to strip leading spaces.

`return String;`
Returns the stripped string.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    Strip_Leading ("  this is a test ... ")
    -- returns "this is a test ... "
```

function Strip_Trailing

```
function Strip_Trailing (From   : String;
                         Filler : Character := ' ') return String;
```

Description

Removes zero or more characters from the end of the specified string.

This function strips trailing characters from a string. Zero or more of the specified characters are stripped from the end of the string. When the last character of the string is not the specified character, the string is returned unchanged.

Parameters

`From :  String;`

Specifies the string from which characters are to be stripped.

`Filler :  Character := ' ';`

Specifies the character to be stripped from the string. The default is to strip trailing spaces.

`return String;`

Returns the stripped string.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    Strip_Trailing ("  this is a test ... ")
    -- returns "  this is a test ..."
```

function Upper_Case

```
function Upper_Case (C : Character) return Character;

function Upper_Case (S : String) return String;
```

Description

Returns the specified character or string in all uppercase characters.

Parameters

`C : Character;`
Specifies the character to be returned in uppercase.

`S : String;`
Specifies the string to be returned in uppercase characters.

`return Character;`
Returns the uppercase character.

`return String;`
Returns the uppercase string.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

procedure Upper_Case

```
procedure Upper_Case (C : in out Character);

procedure Upper_Case (S : in out String);
```

Description

Changes the specified character or string to all uppercase characters.

Parameters

`C : in out Character;`
Specifies the character to be changed to uppercase.

`S : in out String;`
Specifies the string to be changed to uppercase characters.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

end String_Utilities;

generic package Table_Formatter

This package is used to produce neatly formatted tables with centered headers and even amounts of white space between the columns. Columns consist of a title and a series of entries that can be justified left or right or centered. Each entry can be a single item or multiple subitems (see the example below). This package also allows the rows in the table to be sorted on a field or a set of fields.

A table is created through the following sequence of steps:

1. Create a table formatter for the number of columns needed by instantiating the package with the appropriate number of columns.
2. Define the headers for each column and the kind of justification desired for that column by calling the Header procedure once for each column.
3. Enter the items for each column a row at a time. If the entry consists of a single item, call the Item procedure. If the entry comprises multiple subitems, call the Subitem procedure once for each subitem in the item and terminate the entry with a call to the Last_Subitem procedure.
4. Sort the table if desired on a single field or on a set of fields.
5. Print the table that has been defined by calling the Display procedure.

Each instantiation of this package internally allocates enough memory to save a copy of the entire table. Thus it is a good idea to instantiate this package in a local frame so that all the memory it allocates is reclaimed when the frame is discarded.

The formal parameters to the generic are:

```
generic
    Number_Of_Columns  : Positive;
    Subitem_Separator  : String := " ";
package Table_Formatter is
    ...
end Table_Formatter;
```

The Number_Of_Columns parameter defines the number of columns that the table should have. The Subitem_Separator parameter defines the string that should be used to separate the subitems in entries composed of multiple subitems.

Example

The following example illustrates the use of generic package Table_Formatter to create a simple table. The main program is the Print_Table procedure:

```
with Simple_Text_Io;
with Table_Formatter;

procedure Print_Table is

    -- create a table formatter for a 3-column table
    package Table is new Table_Formatter (3);

begin

    -- first define the column headers and the justification
    Table.Header ("Column 1", Table.Left);
    Table.Header ("Column 2", Table.Centered);
    Table.Header ("Column 3", Table.Right);


    -- enter the table entries a row at a time
    Table.Item ("first item");
    Table.Item ("second item");
    Table.Item ("third item");

    Table.Item ("fourth item");

    Table.Subitem ("subitem 1");
    Table.Subitem ("subitem 2");
    Table.Last_Subitem;

    Table.Item ("seventh item");


    -- now display the table
    Table.Display (Simple_Text_Io.Current_Output);

end Print_Table;
```

Calling this procedure causes the following table to be displayed:

```
 Column 1         Column 2         Column 3
===========  ===================  ============
first item       second item        third item
fourth item  subitem 1 subitem 2  seventh item
```

type Adjust

```
type Adjust is (Left, Right, Centered);
```

Description

Defines the types of justification available for column entries.

Enumerations

`Centered`
Causes items to be centered in a column.

`Left`
Causes items to be aligned along the left margin of a column.

`Right`
Causes items to be aligned along the right margin of a column.

References

procedure Header

procedure Display

```
procedure Display (On_File : Io.File_Type);
```

Description

Causes the table that has been created to be written into the specified file.

Parameters

```
On_File :  Io.File_Type;
```

Specifies the file into which the table is to be written. The file must be open as an out file.

Errors

The appropriate exceptions from DIO or TIO, package Io_Exceptions, are raised if the specified file is not open or it is open with the wrong mode.

type Field_List

```
type Field_List is array (Integer range <>) of Positive;
```

Description

Defines an array of field numbers to be used in sorting a table.

Each entry in the array should contain the number of a column of the table on which to be sorted. If multiple columns are supplied, the comparison operation used in sorting compares the items in the first field of the array, and then, if necessary, it continues comparing as many successive fields as necessary to determine if one row is less than another.

References

procedure Sort

procedure Header

```
procedure Header (S      : String;
                  Format : Adjust  := Left);
```

Description

Defines the header for the next column of the table being defined to be the specified string and sets the justification for that column.

The headers of a table are defined sequentially from left to right before items are entered or the table is displayed.

Parameters

`S : String;`

Specifies the header for the column.

`Format :  Adjust := Left;`

Specifies the justification desired for the column.

References

type Adjust

procedure Item

```
procedure Item (S : String);
```

Description

Adds the specified string as the next item in the table being defined.

The entries in the table are entered a row at a time from left to right once the headers have been defined and before the table is displayed.

Parameters

```
S : String;
```

Specifies the item to be entered.

procedure Last_Subitem

```
procedure Last_Subitem;
```

Description

Indicates that the definition of an item composed of multiple subitems is complete.

The entries in the table are entered a row at a time from left to right once the headers have been defined and before the table is displayed. An entry can be a single item or it can be composed of multiple subitems. The subitems for an item are entered from left to right and are terminated by a call to the Last_Subitem procedure. Note that if no subitems are entered, the call to Last_Subitem will cause an empty entry in the table.

References

procedure Subitem

generic formal object Number_Of_Columns

```
Number_Of_Columns : Positive;
```

Description

Defines the number of columns for the table.

procedure Sort

```
procedure Sort (On_Field : Positive := 1);

procedure Sort (On_Fields : Field_List);
```

Description

Sorts the rows of a table that has been defined based on the sort fields supplied.

The sort fields define the columns of the table that should be used in the comparison of rows of the table. If multiple columns are supplied, the comparison operation used in sorting compares the items in the first field of the array, and then, if necessary, it continues comparing as many successive fields as necessary to determine if one row is less than another.

Parameters

`On_Field : Positive := 1;`
Specifies the column on which to sort.

`On_Fields : Field_List;`
Specifies the set of columns on which to sort.

procedure Subitem

```
procedure Subitem (S : String);
```

Description

Adds the specified string as the next subitem of an entry composed of multiple subitems in the table being defined.

The entries in the table are entered a row at a time from left to right once the headers have been defined and before the table is displayed. An entry can be a single item or it can be composed of multiple subitems. The subitems for an item are entered from left to right and are terminated by a call to the Last_Subitem procedure. Note that if no subitems are entered, the call to Last_Subitem will cause an empty entry in the table.

Parameters

```
S : String;
```

Specifies the subitem to be entered.

References

procedure Last_Subitem

generic formal object Subitem_Separator

```
Subitem_Separator : String := " ";
```

Description

Defines the string to be used to separate the subitems for items composed of multiple subitems.

References

procedure Last_Subitem

procedure Subitem

end Table_Formatter;

generic package Unbounded_String

Package Unbounded_String provides a set of operations on objects of the Variable_String type. This type defines a dynamic string with no fixed maximum length. The length of the string is a property of the string itself.

The package performs storage allocation to allow strings to grow dynamically. If strings are no longer needed, their storage can be reclaimed by calling the Free procedure. This package should not be used in a multitasking program unless the user guarantees sequential use of the operations provided.

The package contains several operations that can raise the predefined Constraint_Error exception if improper or illegal input parameter values are provided. Some operations can also raise the predefined Numeric_Error exception if invalid or uninitialized characters are passed to these operations.

The formal parameter to the generic is:

```
generic
    Default_Maximum_Length : Natural := 20;
package Unbounded_String is
    ...
end Unbounded_String;
```

This parameter defines the number of characters in the default storage allocated for new variable strings. This storage is always greater than or equal to the current length of the string and will grow dynamically as the length of the string is increased.

procedure Append

```
procedure Append (Target : in out Variable_String;
                  Source :        Variable_String);

procedure Append (Target : in out Variable_String;
                  Source :        String);

procedure Append (Target : in out Variable_String;
                  Source :        Character);

procedure Append (Target : in out Variable_String;
                  Source :        Character;
                  Count  :        String_Length);
```

Description

Appends the source character or characters to the target string.

This procedure appends characters to the end of an existing variable string. The length of the target variable string is increased by the number of characters in the source variable string. The source can be another variable string, an Ada string, a single character, or a single character appended the number of times specified by the Count parameter.

Parameters

`Target :  in out Variable_String;`

Specifies the object to which characters are to be appended. The length of the variable string is increased by the number of characters in the source.

`Source :  Variable_String;`

Specifies a variable string to be appended to the target.

`Source :  String;`

Specifies an Ada string to be appended to the target.

`Source :  Character;`

Specifies a single character to be appended to the target.

`Count :  String_Length;`

Specifies the number of times the source character is to be appended to the target.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has length Ø
    Append (Command_Name, "Create ");
    -- Command_Name has length 7
    Append (Command_Name, 'x', 3);
    -- Command_Name has length 1Ø
```

function Char_At

```
function Char_At (Source : Variable_String;
                  At_Pos : Positive) return Character;
```

Description

Returns the character at the specified position in the source string.

Parameters

`Source : Variable_String;`

Specifies the variable string whose character is to be found.

`At_Pos : Positive;`

Specifies the position of the desired character in the source.

`return Character;`

Returns the character.

Errors

This function raises the Constraint_Error exception when the At_Pos position is greater than the current length of the source.

The function raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create yyy"
    ... Char_At (Command_Name, 3) -- returns 'e'
```

procedure Copy

```
procedure Copy (Target : in out Variable_String;
                Source :        Variable_String);

procedure Copy (Target : in out Variable_String;
                Source :        String);

procedure Copy (Target : in out Variable_String;
                Source :        Character);
```

Description

Copies the character(s) in the source variable string into the target variable string.

This procedure copies the source character(s) into the target variable string. The target becomes the same length as the source.

Parameters

```
Target :  in out Variable_String;
```

Specifies the variable string in which the copied characters are to be placed. The previous contents of the string are lost. After the copy operation, the current length of this variable string will be the length of the source.

```
Source :  Variable_String;

Source :  String;

Source :  Character;
```

Specifies the source from which characters are to be copied into the target.

Errors

The Numeric_Error exception can be raised if invalid or uninitialized characters are passed to this operation.

Example

```
    Copy (Command_Name, "Create");
```

The six characters of the string are copied into the Command_Name string; the new length of Command_Name is 6. The previous contents of Command_Name are lost.

---

generic formal object Default_Maximum_Length

```
Default_Maximum_Length : Natural := 20;
```

Description

Defines the number of characters in the default storage allocated for new variable strings.

This storage is always greater than or equal to the current length of the string and will grow dynamically as the length of the string is increased.

procedure Delete

```
procedure Delete (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Count  :        String_Length    := 1);
```

Description

Deletes a character or characters from the specified position in the target variable string.

This procedure deletes characters at the At_Pos position from the variable string. The length of the target is decreased by the value of the Count parameter.

Parameters

`Target : in out Variable_String;`

Specifies the variable string from which Count characters are to be deleted.

`At_Pos : Positive;`

Specifies the position in the target at which characters are to be deleted.

`Count : String_Length := 1;`

Specifies the number of characters to be deleted. The default is 1.

Errors

This procedure raises the Constraint_Error exception when the value of the Count parameter is greater than the current length of the target minus the position in the target.

It raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create vvv xxx"
    Delete (Command_Name, 8, 4);
    -- Command_Name has contents "Create xxx"
```

function Extract

```
function Extract (Source    : Variable_String;
                  Start_Pos : Positive;
                  End_Pos   : Natural) return String;
```

Description

Extracts a string from the starting position to the ending position from the source variable string.

This function returns the portion of the variable string that begins with the character at the Start_Pos position and ends with the character at the End_Pos position. When the value of End_Pos is less than the value of Start_Pos, the returned string is null.

Parameters

`Source :  Variable_String;`

Specifies the variable string from which a string is to be extracted.

`Start_Pos :  Positive;`

Specifies the beginning position in the source from which a string is be extracted.

`End_Pos :  Natural;`

Specifies the ending position in the source from which a string is to be extracted.

`return String;`

Returns the extracted string.

Errors

This function raises the Constraint_Error exception when either the Start_Pos or End_Pos value is greater than the current length of the source.

Example

```
    -- Command_Name has contents "Create yyy"
    ... Extract (Command_Name, 8, 10) -- returns "yyy"
```

procedure Free

```
procedure Free (V : in out Variable_String);
```

Description

Reclaims the storage associated with the specified variable string and sets it to the null string.

Parameters

```
V : in out Variable_String;
```

Specifies the string for which storage is to be reclaimed. Upon return the old value will be lost and the string will have the null value.

function Image

```
function Image (V : Variable_String) return String;
```

Description

Converts the variable string into an Ada predefined string.

This function converts an object of the Variable_String type to the String type. If the input has a length of 0, the result is the null string. The result has the same length as the current length of the input and the index range is 1 .. length.

Parameters

```
V : Variable_String;
```

Specifies the object to be converted to the String type.

```
return String;
```

Returns the contents of the variable string in an object of the String type.

procedure Insert

```
procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        Variable_String);

procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        String);

procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        Character);

procedure Insert (Target : in out Variable_String;
                  At_Pos :        Positive;
                  Source :        Character;
                  Count  :        String_Length);
```

Description

Inserts a character or characters from the source variable string into the specified position in the target variable string.

This procedure inserts characters into the target, increasing the length of the target by the number of characters in the source. The characters are inserted between the character at the At_Pos position and the character immediately before that position. The source can be another variable string, an Ada string, a single character, or a single character inserted the number of times specified by the Count parameter.

Parameters

`Target : in out Variable_String;`

Specifies the variable string in which characters from the source are to be inserted.

`At_Pos : Positive;`

Specifies the position at which characters are to be inserted in the target.

`Source : Variable_String;`

Specifies the variable string to be inserted into the target.

`Source : String;`

Specifies the string to be inserted into the target.

```
Source :  Character;
```

Specifies the character to be inserted into the target.

```
Count :  String_Length;
```

Specifies the number of times the character is to be inserted into the target.

Errors

This procedure raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create xxx"
    Insert (Command_Name, 8, "vvv ");
    -- Command_Name has contents "Create vvv xxx"
```

function Is_Nil

```
function Is_Nil (V : Variable_String) return Boolean;
```

Description

Returns true if the indicated string is a null variable string—that is, the variable string returned by the Nil function.

Note that the null string is not the same as the string with the value "".

Parameters

```
V : Variable_String;
```

Specifies the string to be checked.

```
return Boolean;
```

Returns true if the string is null; otherwise, the function returns false.

References

function Nil

function Length

```
function Length  (Source : Variable_String) return String_Length;
```

Description

Returns the current length of the source variable string.

Parameters

`Source :  Variable_String;`

Specifies the variable string whose length is to be found.

`return String_Length;`

Returns the length of the source.

Example

```
    -- Command_Name has contents "Create yyy"
    ... Length (Command_Name) -- returns 10
```

procedure Move

```
procedure Move (Target : in out Variable_String;
                Source : in out Variable_String);
```

Description

Moves the contents of the source variable string to the target variable string, destroying the contents of the source and freeing the storage associated with the source.

This procedure is equivalent to copying the source to the target and then calling the Free procedure on source to reclaim its storage and set its value to the null string.

Parameters

`Target :  in out Variable_String;`

Specifies the variable string into which characters from the source are to be moved. The previous contents are destroyed.

`Source :  in out Variable_String;`

Specifies the variable string from which characters are to be moved to the target. Moving the contents to the target destroys them in the source, reclaiming the storage there. The value of the variable string following the operation is the null string.

function Nil

```
function Nil return Variable_String;
```

Description

Returns the null string.

Note that the null string is not the same as the string with the value "".

Parameters

```
return Variable_String;
```

Returns the null string.

References

function Is_Nil

procedure Replace

```
procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        Character);

procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        Character;
                   Count  :        String_Length);

procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        String);

procedure Replace (Target : in out Variable_String;
                   At_Pos :        Positive;
                   Source :        Variable_String);
```

Description

Replaces character(s) in the target variable string with character(s) from the source variable string.

This procedure replaces characters in the target with characters from the source. The length of the target remains the same. The number of characters replaced is the number of characters in the source. The source can be another variable string, an Ada string, a single character, or a single character repeated the number of times specified by the Count parameter.

Parameters

`Target : in out Variable_String;`

Specifies the variable string in which characters from the source are to be replaced.

`At_Pos : Positive;`

Specifies the position at which characters are to be replaced in the target.

`Source : Character;`

Specifies the character to be replaced in the target.

`Source : String;`

Specifies the Ada string to be replaced in the target.

```
Source : Variable_String;
```
Specifies the variable string to be replaced in the target.

```
Count : String_Length;
```
Specifies the number of times the character is to be replaced in the target.

Errors

This procedure raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create xxx"
    Replace (Command_Name, 8, "yyy");
    -- Command_Name has contents "Create yyy"
```

procedure Set_Length

```
procedure Set_Length (Target     : in out Variable_String;
                      New_Length :        String_Length;
                      Fill_With  :        Character        := ' ');
```

Description

Truncates or fills the variable string to the specified length.

This procedure sets the length of the target variable string to the value of the New_Length parameter. When the new length is less than the old length, the variable string is truncated. When the new length is the same as the old length, the procedure has no effect. When the new length is greater than the old length, the variable string is extended with the new character positions filled with the Fill_With parameter.

Parameters

`Target :  in out Variable_String;`

Specifies the variable string whose length is to be changed.

`New_Length :  String_Length;`

Specifies the new length of the target.

`Fill_With :  Character := ' ';`

Specifies the fill character. This character is significant only if the target is made larger. The default is a space character.

Errors

This procedure raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

Example

```
    -- Command_Name has contents "Create yyy"
    Set_Length (Command_Name, 12);
    -- Command_Name has contents "Create yyy  "
```

subtype String_Length

```
subtype String_Length is Natural;
```

Description

Defines the allowed values of the length of the variable string.

The length of the variable string can be between 0 and Integer'Last, inclusive.

function Value

```
function Value (S : String) return Variable_String;
```

Description

Converts the string into a variable string.

This function converts an object of the String type to the Variable_String type. The result has the same length as the length of the string.

Parameters

```
S : String;
```

Specifies the object to be converted to the Variable_String type.

```
return Variable_String;
```

Returns the contents of the input string in an object of the Variable_String type.

type Variable_String

```
type Variable_String is private;
```

Description

Defines the representation of a variable string.

Several important properties of the Variable_String type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of a variable string are not copied but a new alias (a new name) for the string is created. Equality for this type has the property that the values of the strings are not compared but the names are compared. In other words, the operation of equality checks to determine whether any two variable string values designate the same variable string.

end Unbounded_String;

RATIONAL

Index

This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

RATIONAL

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

____
____
____
____
____
____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

____
____
____
____

Indicate any additions or changes you would like to see in the index.

____
____
____
____

How much experience have you had with the Rational Environment?

6 months or less ______ 1 year ______ 3 years or more ______

How much experience have you had with the Ada programming language?

6 months or less ______ 1 year ______ 3 years or more ______

Name (optional)______________________________ Date______________
Company ____
Address ____
City ______________________ State ______________ ZIP Code ____________

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043