Section 6 - UIL Data Types

This page describes the format and contents of each reference page in Section 6, which covers each of the UIL data types.

Name

Type – a brief description of the data type.

Synopsis

Syntax:

The literal syntax for specifying a value of the data type. Anything in constant width type should be typed exactly as shown. Items in italics are expressions that should be replaced by actual values when you specify a value. Anything enclosed in brackets is optional. An ellipsis (...) means that the previous expression can be repeated multiple times and a vertical bar (|) means to select one of a set of choices.

MrmType:

The Mrm value type that corresponds to the data type. These types are returned by MrmFetchLiteral().

Availability

This section appears for data types that were added in Motif 2.0 or later.

Description

This section gives an overview of the data type. It explains the literal syntax that is used to specify a value of the type in a UIL module.

The UIL compiler supports *integer, float*, *single_float, boolean*, *string,* and *compound_string* expressions in most contexts where a value of one of the types is expected. Expressions can include literal or named values, but any named values that are used must be declared private *or exported* because the result of an expression cannot be computed if it contains an imported *value.*

The UIL compiler allows both string and arithmetic expressions. String expressions contain NULL-terminated strings and compound strings, while arithmetic expressions can contain integer, float, single_float, and boolean values.

A string expression, consists of two or more string or compound_string values concatenated with the string concatenation operator (&). The string and compound_string reference sections contains more details and examples of string concatenation.

An arithmetic expression consists of one or more boolean, integer, single_float, or float values and one or more arithmetic operators. The following operations can be used in arithmetic expressions:

Operator	Type	Operand Types	Operation	Precedence
~	unary	boolean	NOT	1 (highest)
		integer	One’s complement	1
-	unary	integer	Negation	1
		float	Negation	1
+	unary	integer	None	1
		float	None	1
*	binary	integer	Multiplication	2
		float	Multiplication	2
/	binary	integer	Division	2
		float	Division	2
+	binary	integer	Addition	3
		float	Addition	3
-	binary	integer	Subtraction	3
		float	Subtraction	3
>>	binary	integer	Shift right	4
<<	binary	integer	Shift left	4
&	binary	boolean	AND	5
		integer	Bitwise AND	5
|	binary	boolean	OR	6
		integer	Bitwise OR	6
^	binary	boolean	XOR	6
		integer	Bitwise XOR	6 (lowest)

When the UIL compiler evaluates an expression, higher precedence operations are performed before those of lower precedence. Binary operations of equal precedence are evaluated from left to right, while unary operations of equal precedence are evaluated from right to left. You can change the default order of evaluation by using parentheses to group subexpressions that should be evaluated first. For example, in the expression 2+4*5, 4*5 is evaluated first, followed by 20+2. If the expression is written (2+4)*5, then 2+4 is evaluated first, followed by 6*5.

The type of an expression is the type of its most complex operand. The UIL compiler converts the value of the less complex type in an operation to a value of the most complex type. The order of complexity for operands in a string expression is string followed by compound_-string. For operations in an arithmetic expression, the order is boolean, integer, single_float, and float.

For example, if a string expression contains only strings, the type of the concatenated expression is string, but if it contains both strings and compound strings, its type is compound_-string. The result of concatenating two NULL-terminated strings is a NULL-terminated string, unless the two strings have different character sets or writing directions, in which case the result is a compound string. If an arithmetic expression contains only integers, the type of an expression is integer, but if it contains both integers and floats, its type is float.

The table below summarizes the valid uses of the types documented in this section. For each type, the table indicates the supported storage classes. It also specifies whether or not values of the type can be specified literally and whether or not the type can be used for a procedure parameter and as an argument type. The final column lists the Motif Resource Manager (Mrm) routine that can be used to fetch values of the type. If certain information is not relevant for a type, the table entry indicates that it is not applicable (NA).

	Supported Storage Classes					
Type	**Private**	**Exported**	**Imported**	**Literal Value**	**Reason/ Parameter**	**Fetch Function**
any	NA	NA	NA	No	Yes	NA
argument	Yes	No	No	Yes	No	NA
asciz_table	Yes	Yes	Yes	Yes	Yes	MrmFetchLiteral
boolean	Yes	Yes	Yes	Yes	Yes	MrmFetchLiteral
character_set	NA	NA	NA	Yes	No	NA
class_rec_name	Yes	Yes	Yes	Yes	Yes	MrmFetchLiteral
color	Yes	Yes	Yes	Yes	Yes	MrmFetchColorLiteral
color_table	Yes	No	No	Yes	No	NA
compound_string	Yes	Yes	Yes	Yes	Yes	MrmFetchLiteral
compound_string_component	Yes	Yes	Yes	Yes	Yes	MrmFetchLiteral

Type	Supported Storage Classes			Literal Value	Reason/ Parameter	Fetch Function
	Private	Exported	Imported			
compound_string _table	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
float	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
font	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
fontset	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
font_table	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
icon	Yes	Yes	Yes	Yes	Yes	MrmFetch-IconLiteral, MrmFetchBit-mapLiteral
integer	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
integer_table	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
keysym	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
pixmap	No	No	Yes	No	Yes	NA
reason	Yes	No	No	Yes	No	NA
rgb	Yes	Yes	Yes	Yes	Yes	MrmFetch-ColorLiteral
single_float	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
string	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
translation_table	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral
wide_character	Yes	Yes	Yes	Yes	Yes	MrmFetchLit-eral

Type	Supported Storage Classes			Literal Value	Reason/ Parameter	Fetch Function
	Private	Exported	Imported			
widget	Yes	Yes	Yes	Yes	Yes	MrmFetch-Widget, MrmFetch-WidgetOver-ride
xbitmapfile	Yes	Yes	Yes	Yes	Yes	MrmFetch-IconLiteral

The UIL compiler may not generate errors when some of the types are used incorrectly. These cases are documented in the individual type reference pages.

As of Motif version 1.2, the UIL compiler does not support the assignment of a character_set value to a named variable. A built-in or literal character set must be specified in all contexts in which a character set is expected. In addition, prior to Motif 1.2.1, UIL may generate an error if the type widget is used as an argument or reason type. In this case, the type any can be used as a workaround.

Usage

This section provides less formal information about the data type: when and how you might want to use it and things to watch out for.

Example

This section provides examples of the use of the type.

See Also

This section refers you to related functions, UIL file format sections, and UIL data types. The numbers in parentheses following each reference refer to the sections of this book in which they are found.

Name

any – type checking suppression type.

Synopsis

Syntax:

any

MrmType:

MrmRtypeAny

Description

The *any* type is used to suppress type checking for values passed to callback procedures or assigned to user-defined arguments. When a callback parameter or user defined-argument type is specified as *any*, the UIL compiler allows a value of any type to be used. Because the type *any* is only used to specify an expected type in these two cases, it does not have a literal syntax and values of type *any* cannot be defined or declared.

Usage

The *any* type specifier is used when values of more than one type can be passed as a callback parameter or assigned to an argument. It can also be used when a callback or argument expects a type that is not predefined by the UIL compiler.

Since no type checking is performed on callback parameters or arguments declared as type *any*, it is possible to specify a value that is not expected by the callback or widget. You should use caution when specifying the value for a callback or argument that uses the any type.

Example

```
...
! Define activate procedure that takes different arguments depending upon
! usage context. Context must be checked in C code before value is used.
procedure
    activate (any);
! Define a resource that can be set to different types.
! Widget checks type field at run-time to determine value type.
value
    XtNlabelValue    : argument ('labelValue', any);
    XtNlabelType     : argument ('labelType', integer);
```

See Also

`procedure`(5), `argument`(6).

Name

argument – user-defined resource type.

Synopsis

Syntax:

argument (*string_expression* [, *argument_type*])

MrmType:

none

Description

An *argument* value represents a user-defined resource. An *argument* is represented literally by the symbol *argument*, followed by a string expression that evaluates to the name of the resource and an optional resource type. The name of the resource is assigned to the name member of the ArgList structure passed to `XtSetValues`(). The name is typically the name of a resource with the XmN or XtN prefix removed. The type of the argument, if specified, is used by the UIL compiler to perform type checking of assignments to the resource. If omitted, the type defaults to any.

Usage

A user-defined resource can be used in the *arguments* section of a UIL module, for both built-in Motif widgets and user-defined widgets. While user-defined arguments are typically assigned to a named variable in the value section, they can also be specified literally in the arguments section of an *object* definition. If you are defining arguments for a widget or widget set that is not predefined, you should define them as named variables in a separate UIL module that can be included by any module that uses the widget(s).

Arguments must be private values; they cannot be imported or exported. The UIL compiler allows imported and exported declarations, but it generates an error when the user-defined argument is used. Since argument values cannot be exported, they cannot be retrieved by an application.

The argument type can only be used to define non-callback resource types. The *reason* type is used to specify user-defined callback resources.

Some versions of the UIL compiler may not allow the definition of arguments of type widget. If you encounter this problem, use the type any as a workaround. The compiler may allow the definition of arguments of type *argument* or *reason*. If arguments with these types are used, the actual value set as the widget's resource is undefined.

Example

From *Xaw/Tree.uih*:

```
! Resource and definitions for the Athena Tree widget.
value
    XtNautoReconfigure : argument (‘autoReconfigure’, boolean);
    XtNgravity         : argument (‘gravity’, integer);
    NorthGravity       : 2;
    WestGravity        : 4;
    EastGravity        : 6;
    SouthGravity       : 8;
    ! Use any type because compiler may not allow widget:
    XtNtreeParent : argument (’treeParent’, any);
...
```

From *my_module.uil*:

```
include file ’Xaw/Tree.uih’;
object parent : XmPushButton { }
object child : XmPushButton {
    arguments {
        XtNtreeParent = parent;
    };
};

object tree : procedure user_defined XawCreateTreeWidget {
    arguments {
        XtNautoReconfigure = false;
        XtNgravity = NorthGravity;
    };
    controls {
        XmPushButton parent;
        XmPushButton child;
    };
};
```

See Also

`MrmRegisterClass`(3), `include`(5), `object`(5), `reason`(6).

Name

asciz_string_table – array of NULL-terminated strings.

Synopsis

Syntax:

asciz_table (*string_expression* [, ...]) or
asciz_string_table (*string_expression* [, ...])

MrmType:

MrmRtypeChar8Vector

Description

An *asciz_string_table* value represents an array of NULL-terminated strings. An *asciz_string_table* is represented literally by the symbol *asciz_table* or *asciz_string_table*, followed by a list of string expressions separated by commas. String variables in this list can be forward referenced.

Usage

There are no built-in Motif resources of type *asciz_string_table*, so values of this type are usually passed as callback parameters or retrieved with `MrmFetchLiteral`(). The type *asciz_string_table* can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal. An *asciz_string_table* obtained by the application as a callback parameter, a widget resource, or with `MrmFetchLiteral`() is NULL-terminated.

Example

```
...
! Declare a procedure that expects an array of NULL-terminated strings.
procedure
    set_names (asciz_table);

! Define a couple of asciz_tables
value
    dwarfs      : asciz_table (‘Dopey’, ‘Doc’, ‘Sneezy’, ‘Sleepy’, ‘Happy’,
    ’Grumpy’, ’Bashful’);
    numbers     : asciz_string_table (one, two);
    one         : ’one’;
    two         : ’two’;
    reindeer    : imported asciz_string;

! Define some asciz_table resources.
value
    XtNniceList       : argument (‘niceList’, asciz_table);
    XtNnaughtyList   : argument ('naughtyList', asciz_table);
object doit : XmPushButton {
    callbacks {
        XmNactivateCallback = procedure set_names (dwarfs);
    };
};
...
```

See Also

`MrmFetchLiteral`(3), `procedure`(5), `argument`(6), `compound_string`(6), `compound_string_table`(6), `string`(6).

Name

boolean – true/false type.

Synopsis

Syntax:

true | on | false | off

MrmType:

MrmRtypeBoolean

Description

Values of type *boolean* may be either true (on) or false (off). A *boolean* value is represented literally by *true, false, on,* or *off*. A *boolean* variable can be defined in the value section by setting a named variable to one of these literal values or to another boolean variable.

Usage

The type name *boolean* can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an *argument* literal.

A boolean value can be explicitly converted to an *integer*, *float*, or *single_float* value by specifying the conversion type followed by the *boolean* value in parentheses. *true* and *on* convert to the value 1 or 1.0, while *false* and *off* convert to the value 0 or 0.0.

The storage allocated by Mrm for a *boolean* value is sizeof(int) not sizeof(Boolean). Because sizeof(Boolean) is less than sizeof(int) on many systems, you should use an int pointer rather than a Boolean pointer when retrieving a boolean value with `MrmFetchLiteral`().

Example

```
...
procedure
    set_sleepy_state (boolean);

value
    map_flag    : true;
    one         : integer (true);
    zero        : integer (false);
    debug       : imported boolean;
    XtNtimed    : argument ('timed', boolean);

object sleep : XmPushButton {
    arguments {
        XmNmapWhenManaged = map_flag;
```

```
            XmNtraversalOn = off;
        };
        callbacks {
            XmNactivateCallback = procedure set_sleepy_state (true);
        };
    };
    ...
```

See Also

`MrmFetchLiteral`(3), `procedure`(5), `argument`(6), `float`(6), `integer`(6), `single_float`(6).

Name

character_set – character set type for use with strings and font lists.

Synopsis

Syntax:

```
character_set ( string_expression
                              [, right_to_left = boolean_expression ]
                              [, sixteen_bit = boolean_expression ] )
```

MrmType:

none

Description

The *character_set* type represents a user-defined character set that can be used when defining *strings*, *compound_strings*, *fonts*, *fontsets*, and *font_tables*. A character set specifies the encoding that is used for character values. A *character_set* is represented literally by the symbol *character_set*, followed by a string expression that names the character set and two optional properties.

If the *right_to_left* property of the character set for a string is set to *true*, the string is parsed and stored from right to left and compound strings created from the string have a direction component of XmSTRING_DIRECTION_R_TO_L. The default value of this property is *false*. The direction component used by a compound_string can be specified independently of the parsing direction using the compound_string literal syntax.

If the *sixteen_bit* property of the character set for a string is set to *true*, the string is interpreted as having double-byte characters. Strings with this property set to true must contain an even number of bytes or the UIL compiler generates an error.

Usage

A *character_set* value is used to specify the character set for *string*, *compound_string*, *font*, *fontset*, and *font_table* values. The *right_to_left* and *sixteen_bit* properties only apply to strings and compound strings and have no effect on character sets specified for fonts and fontsets.

Unlike most of the UIL types, the *character_set* type cannot be assigned to a named variable in a *value* section, or used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal. A character set value can only be specified with the *character_set* literal syntax.

If a *font*, *fontset*, or *font_table* that uses a user-defined character set is exported or used as a resource value, the UIL compiler may exit with a severe internal error. As a result, only the predefined character sets can be used with *font*, *fontset*, and *font_list* values. You can work around this problem by specifying values of these types in an X resource file.

The UIL compiler may allow the use of string variables and the string concatenation operator (&) in a *character_set* name specification. Although no errors are generated, a string using such a character set may be incorrectly converted to a *compound_string* value. To avoid this problem, you should always specify a quoted string as the name in a *character_set* literal.

UIL defines a number of built-in character sets that you can use to define *string*, *compound_string*, *font*, *fontset*, and *font_table* values. The following table summarizes the built-in character sets:

UIL Name	**Character Set**	**Parse Direction**	**Writing Direction**	**16 Bit**
iso_latin1	ISO8859-1	L to R	L to R	No
iso_latin2	ISO8859-2	L to R	L to R	No
iso_latin3	ISO8859-3	L to R	L to R	No
iso_latin4	ISO8859-4	L to R	L to R	No
iso_latin5	ISO8859-5	L to R	L to R	No
iso_cyrillic	ISO8859-5	L to R	L to R	No
iso_arabic	ISO8859-6	L to R	L to R	No
iso_arabic_lr	ISO8859-6	L to R	R to L	No
iso_greek	ISO8859-7	L to R	L to R	No
iso_hebrew	ISO8859-8	R to L	R to L	No
iso_hebrew_lr	ISO8859-8	L to R	R to L	No
jis_katakana	JISX0201.1976-0	L to R	L to R	No
gb_hanzi	GB2313.1980-0	L to R	L to R	Yes
gb_hanzi_gr	GB2313.1980-1	L to R	L to R	Yes
jis_kanji	JISX0208.1983-0	L to R	L to R	Yes
jis_kanji_gr	JISX0208.1983-1	L to R	L to R	Yes
ksc_hangul	KSC5601.1987-0	L to R	L to R	Yes
ksc_hangul_gr	KSC5601.1987-1	L to R	L to R	Yes

Example

```
...
value
    ! Define font with user-defined character set.
    big: font ('*times-medium-r-normal-*-240-75-75-*',
    character_set = character_set ('body'));
    ! Declare some strings with user-defined character sets.
    player : #character_set (big) "Mookie Wilson";
    hello : exported #iso_hebrew "\355\345\354\371\";
    ...
```

See Also

`compound_string`(6), `font`(6), `fontset`(6), `font_table`(6), `string`(6).

Name

class_rec_name – widget class pointer type.

Synopsis

Syntax:

class_rec_name (*string_expression)*

MrmType:

MrmRtypeClassRecName

Description

The *class_rec_name* type represents a pointer to a widget class record. A class_rec_name value is represented literally by the symbol *class_rec_name*, followed by a string that specifies the class name. The string can either be the name of a class from a widget's class definition or the name of a widget creation function registered with `MrmRegisterClass`(). The string is converted to a widget class pointer at run-time by Mrm when a *class_rec_name* value is referenced. Mrm finds the widget class pointer corresponding to the name by searching the list of widgets registered with `MrmRegisterClass`(). This list includes the built-in Motif widgets and any user-defined widgets that have been registered.

Usage

The type *class_rec_name* can be used as the type of an imported value, as the parameter type in a procedure declaration, or as the type in an argument literal. None of the built-in Motif widgets have a class_rec_name resource, however. If a *class_rec_name* value is specified as a resource value for a widget and the conversion of the class name string to a widget class pointer fails at run-time (inside a call to `MrmFetchWidget`(), `MrmFetchWidgetOverride`(), or `MrmFetchSetValues`()), Mrm does not set the resource. If `MrmFetchLiteral`() is used to retrieve the value and the conversion fails, MrmNOT_FOUND is returned.

Example

```
...
value
    pbclass : class_rec_name ('XmPushButton');
...
```

See Also

`MrmFetchSetValues`(3), `MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3), `MrmInitialize`(3), `MrmRegisterClass`(3), `procedure`(5), `argument`(6).

Name

color – color specified as color name.

Synopsis

Syntax:

color (*string_expression* [foreground | background])

MrmType:

MrmRtypeColor

Description

A *color* value represents a named color. A color is represented literally by the symbol *color*, followed by a string expression that evaluates to the color name and an optional foreground or background property to indicate how the color is displayed on a monochrome screen. Mrm converts the color name to an X Color at run-time with `XAllocNamedColor`() on a color display, or chooses black or white on a monochrome display. The X server maintains a color name database that is used to map color names to RGB values. The text version of this database is typically in the file */usr/lib/x11/rgb.txt.* See Volume One, *Xlib Programming Manual,* and Volume Two, *Xlib Reference Manual,* for more information on color allocation.

Usage

The *color* type can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal. An *rgb* value can also be specified in any context that a color value is valid. There are several built-in Motif *color* resources, such as XmNforeground and XmNbackground.

The optional *foreground* and *background* properties can be used to specify the mapping of colors on a monochrome display or when a color allocation fails because the colormap is full. Mrm dynamically determines the appropriate foreground or background color based on the context in which a *color* value is used.

When a *color* is used as a resource value for a widget (directly or indirectly in an icon's *color_table*), the background and foreground colors are obtained from the widget. When a color is retrieved for the *color_table* of an icon retrieved with `MrmFetchIconLiteral`(), the background and foreground colors are supplied by the application as arguments to the function.

If the *foreground* or *background* property is not specified, Mrm uses the Color returned by `XAllocNamedColor`() on a monochrome display. When an allocation fails on a color display and neither property is specified, black is used. In addition, black is always used when an allocation on a color display fails in

`MrmFetchColorLiteral`(); the procedure does not take fallback background and foreground colors arguments.

As of Motif version 1.2.1, the color substitutions described above do not take place. When a color allocation fails for a color specified directly or indirectly as a resource value, the resource is not set. If the allocation fails in a call to `MrmFetchColorLiteral`() or `MrmFetchIconLiteral`(), MrmNOT_FOUND is returned.

Example

```
...
value
    background : color ('chocolate mint', background);
    foreground : color ('whipped cream', foreground);

object label: XmLabel {
    arguments {
        XmNbackground = color ('red');
    };
};
...
```

See Also

`MrmFetchColorLiteral`(3), `MrmFetchIconLiteral`(3), `MrmFetchSetValues`(3), `MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3), `color_table`(6), `icon`(6), `rgb`(6).

Name

color_table – character-to-color mapping type.

Synopsis

Syntax:

color_table (*color_expression* = ’*character’* [, ...])

MrmType:

none

Description

A *color_table* value is used to define a mapping from color names or RGB values to the single characters that are used to represent pixel values in icons. A color_table is represented literally by the symbol *color_table*, followed by a list of mappings. Each mapping associates a previously-defined color value with a single character. A color value can be a variable or a literal of type *color* or *rgb*, the global *background color*, or the symbol *foreground color*.

Usage

The sole purpose of a *color_table* is to define colors that can be used in an icon definition. Because the color mappings are needed at compile-time to construct an icon, a *color_table* value must be private. The UIL compiler may allow an imported or exported *color_table* definition, but it generates an error when the value is used. Unlike most other UIL types, a *color_table* cannot be used as a parameter type in a procedure declaration or as the type in an argument literal.

The *color* values *background color* and *foreground color* can be used to map a character to the background or foreground color. These colors are determined at run-time by Mrm, based on the context in which an *icon* is used. When an *icon* is a resource value for a widget, the foreground and background colors are obtained from the widget. When an *icon* is retrieved by the application with `MrmFetch-IconLiteral`(), the foreground and background colors are supplied by the application as arguments to the function.

The colors in a *color_table* are allocated at run-time by Mrm when an icon that uses the color table is retrieved as the value for a widget resource or retrieved by the application with `MrmFetchIconLiteral`(). See the *color* reference page for a description of how Mrm allocates colors and what happens when a color allocation fails.

The UIL compiler may not perform type checking on the color values in a *color_table*. If the compiler allows the use of a value that is not a color, it will crash when the *color_table* is used.

Example

```
...
value
    blue : color ('blue');
    yellow : rgb (65535,65535,0);
    palette : color_table (     background color = ' ',
                                foreground color = '*',
                                color ('red') = 'r',
                                rgb (0,65535,0) = 'g',
                                blue = 'b',
                                yellow = 'y');

plus : icon (color_table = palette, 'brb', 'rrr', 'brb');
...
```

See Also

`MrmFetchIconLiteral`(3), `color`(6), `icon`(6), `rgb`(6).

Name

compound_string – Motif compound string type.

Synopsis

Syntax:

compound_string (*string_expression*
    [, character_set = *character_set*]
    [, right_to_left = *boolean_expression*]
    [, separate = *boolean_expression*])

MrmType:

MrmRtypeCString

Description

A *compound_string* value represents a Motif XmString. An XmString is the data type for a Motif compound string. The Motif toolkit uses compound strings, rather than character strings, to represent most text values. A compound string is composed of one or more segments, where each segment can contain a font list element tag, a string direction, and a text component. The tag specifies the font, and thus the character set, that is used to display the text component.

UIL-generated *compound_strings* can contain up to four components: a single-byte, multi-byte, or wide-character string, a character set, a writing direction, and a separator. Like NULL-terminated strings, *compound_strings* can be concatenated with the concatenation operator (&). A *compound_string* is represented literally by the symbol *compound_string*, followed by a string expression and an optional list of properties. The valid properties are *character_set*, *right_to_left*, and *separate*. They may be specified in any order, but each may occur only once.

The character_set property is used to establish the character set of the *compound_string*. It can be set to one of the UIL built-in character sets or to a user-defined character set. If a character set is specified in the definition of the string using the #*character_set* notation, it takes precedence over the character_set property setting. If the *character_set* property is omitted, the default character set of the module is used.

The *right_to_left* property is used to set the writing direction of the *compound_string*. If the *right_to_left* property is omitted, the writing direction defaults to that of the character set of the *compound_string*.

When the *separate* property is set to true, UIL adds a separator component to the end of the *compound_string*. Separators usually appear as line breaks when a compound string is displayed. If omitted, the separate property defaults to false. Newline characters present in the string expression of a compound string literal are not converted to separators.

Usage

When a *compound_string* literal contains a string expression consisting of two or more concatenated strings, they are combined into a single component if the character set and writing direction of each is the same. If any of the character sets differ, each string is placed in a separate string component with its own character set and direction components. If the separate property is set to true, a separator component is added to the end of the entire compound_string.

A *compound_string* with a character_set that differs from XmFALLBACK_-CHARSET is only displayed correctly in a Motif widget if the XmFontList of the widget includes an XFontStruct or an XFontSet entry for the character_set.

The type *compound_string* can also be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an *argument* literal.

Example

```
...
procedure
    set_label_string (compound_string);

value
    ying  : "Ying";
    yang  : #iso_latin1"Yang";
    left  : compound_string (ying, character_set=iso_latin1, separate=true);
    right : compound_string (yang, right_to_left=true);
    day   : compound_string ('moon' & ' ' & 'sun');
    other : imported compound_string;
lines : exported left & right;

object verse : XmLabel {
    arguments {
        XmNlabelString = lines;
    };
};

value
    XtNgraphicCaption : argument ('graphicCaption', compound_string);
...
```

See Also

`XmStringCreate`(1), `XmStringCreateLocalized`(1), `character_set`(6), `compound_string_component`(6), `compound_string_table`(6), `string`(6).

Name

compound_string_component – Motif compound string component type.

Synopsis

Syntax:

compound_string_component (*component_type* [, { *string* | *enumval* }])

MrmType:

MrmRtypeCString

Availability

Motif 2.0 and later.

Description

A *compound_string_component* value represents a compound string containing a single component. It is the UIL equivalent of the Motif function `XmStringComponentCreate`(). The compound string so produced can be concatenated with other segments to create more complex compound strings. As for the *compound_string* data type, the symbol & is the concatenation operator.

The *component_type* parameter specifies the type of compound string segment to be created. The value is one of the constants defined for the XmStringComponentType enumeration. Depending upon the type of the segment, a second qualifying parameter may be required: the valid component types, together with any extra argument is as follows:

Where *component_type* is XmSTRING_COMPONENT_DIRECTION, *compound_string_component* is equivalent to the Motif function `XmStringDirectionCreate`(), and the XmStringDirection argument is as required for that function: XmSTRING_DIRECTION_L_TO_R, XmSTRING_DIRECTION_R_TO_L, or XmSTRING_DIRECTION_DEFAULT.

Usage

The *compound_string_component* data type can be used in an analogous fashion to the *compound_string* type. The differences lie in the degree of control in constructing the compound strings: tab, separator, and rendition components can be created. The components XmSTRING_COMPONENT_RENDITION_BEGIN and XmSTRING_COMPONENT_RENDITION_END take as argument a string which is matched against a rendition tag within the current render table.

Example

```
...
value
      tab            : compound_string_component
                              (XmSTRING_COMPONENT_TAB);
      separator      : compound_string_component
                              (XmSTRING_COMPONENT_SEPARATOR);
      charset        : compound_string_component
                              (XmSTRING_COMPONENT_CHARSET,
                              iso_latin1);
      l_to_r_text    : compound_string_component
                              (XmSTRING_COMPONENT_TEXT,
                              "left_to_right");
      r_to_l_text    : compound_string_component
                              (XmSTRING_COMPONENT_TEXT,
                              "left-to-right");
      r_to_l         : compound_string_component
                              (XmSTRING_COMPONENT_DIRECTION,
                              XmSTRING_DIRECTION_R_TO_L);
      l_to_r         : compound_string_component
                              (XmSTRING_COMPONENT_DIRECTION,
                              XmSTRING_DIRECTION_L_TO_R);
      cstring        : r_to_l & charset & r_to_l_text & separator & l_to_r &
                              l_to_r_text;

object label: XmLabel {
      arguments {
            XmNlabelString = cstring;
      }
};
...
```

See Also

`XmStringComponentCreate`(1), `XmStringDirectionCreate`(1), `compound_string`(6), `compound_string_table`(6), `string`(6).

Name

compound_string_table – array of compound strings.

Synopsis

Syntax:

compound_string_table (*string_expression* [, ...]) or
string_table (*string_expression* [, ...])

MrmType:

MrmRtypeCStringVector

Description

A *compound_string_table* value represents an array of Motif XmStrings. An XmString is the data type for a Motif compound string. The Motif toolkit uses compound strings, rather than character strings, to represent most text values. A compound string is composed of one or more segments, where each segment can contain a font list element tag, a string direction, and a text component. The tag specifies the font, and thus the character set, that is used to display the text component.

A *compound_string_table* is represented literally by the symbol *compound_string_table* or *string_table*, followed by a list of *string* or *compound_string* expressions. The UIL compiler automatically converts a string expression to a *compound_string*.

Usage

A common use of *compound_string_table* values is to set resources of the type XmStringTable in a UIL module or in the application with `MrmFetchSetValues`(). When a *compound_string_table* is assigned to a built-in XmStringTable resource, UIL automatically sets the corresponding count resource. The table below lists the XmStringTable resources and their related count resources.

Widget	**XmStringTable Resource**	**Related Resource**
XmList	XmNitems	XmNitemCount
XmList	XmNselectedItems	XmNselectedItemCount
XmSelectionBox	XmNlistItems	XmNlistItemCount
XmCommand	XmNhistoryItems	XmNhistoryItemCount
XmFileSelectionBox	XmNdirListItems	XmNdirListItemCount
XmFileSelectionBox	XmNfileListItems	XmNfileListItemCount

The associated count is not automatically set for compound_string_tables that are assigned using `MrmFetchSetValues`().

The type *compound_string_table* can also be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal. A *compound_string_table* that is obtained by the application as a callback parameter, a widget resource, or with `MrmFetchLiteral`() is NULL-terminated.

If a *compound_string_table* contains a forward reference to a *compound_string* value, all items in the list before that entry may be lost by the UIL compiler. To avoid this problem, you should be sure to define all compound_strings used in a *compound_string_table* before they are referenced.

Example

```
...
procedure
    set_items (string_table);

value
    fruit_list : string_table ('apple', 'banana', 'grape');

object list : XmList {
    arguments {
        XmNitems = fruit_list;
    };
};

value
    XtNnameList : argument ('nameList', compound_string_list);
...
```

See Also

`character_set`(6), `compound_string`(6), `compound_string_component`(6), `string`(6).

Name

float – double-precision floating point type.

Synopsis

Syntax:

[+ | -]*integer.integer* [e [+ | -]*integer*]

MrmType:

MrmRtypeFloat

Description

A *float* value represents a negative or positive double-precision floating point number. A *float* is represented literally by an optional sign, one or more consecutive digits which must include a decimal point, and an optional exponent. The UIL compiler uses `atof`() to convert literal float values to the architecture's internal representation.

A float can also be represented literally by the symbol *float* followed by a *boolean*, *integer*, or *single_float* expression. The expression is converted to a float and can be used in any context that a *float* value is valid. A float is formed from a boolean by converting true and on to 1.0 and false and off to 0.0.

Usage

The allowable range of a *float* value is determined by the size of a C double on the machine where the UIL module is compiled. Since a double on most architectures is typically a minimum of four bytes, float values may safely range from 1.4013e-45 to 3.40282e+38 (positive or negative). Although many architectures represent a double using eight bytes, you can ensure greater portability by keeping *float* values within the four-byte range. The UIL compiler generates an error if it encounters a *float* outside of the machine's representable range.

The type *float* can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal.

Example

```
...
! Declare some floating point values.
value
    pi              : 3.14159;
    burn_rate       : imported float;
    one_point_oh    : float (true);
    ten_even        : float (10);
```

```
! Declare a procedure which takes a float parameter.
procedure
    set_temperature (float);

! Declare an argument of type float.
value
    XtNorbitalVelocity : argument ('orbitalVelocity', float);
...
```

See Also

`boolean`(6), `integer`(6), `single_float`(6).

Name

font – XFontStruct type.

Synopsis

Syntax:

font (*string_expression* [, character_set = *character_set*])

MrmType:

MrmRtypeFont

Description

A *font* value represents an XFontStruct, which is an Xlib structure that specifies font metric information. A font is represented literally by the symbol font, followed by a string expression that evaluates to the name of the font and an optional character_set. All parts of the string expression that make up the font name must be private to the UIL module. The *character_set* is associated with the font if it appears in a font_table. If *character_set* is not specified, it is determined from the codeset portion of the LANG environment variable if it is set, or XmFALLBACK_CHARSET otherwise.

The string expression that specifies the font name is an X Logical Font Description (XLFD) string. This string is stored in the UID file and used as a parameter to `XLoadQueryFont()` at run-time to load the font. See Volume One, *Xlib Programming Manual*, and Volume Two, *Xlib Reference Manual*, for more information on fonts.

Usage

You can use a *font* value to specify a font or *font_table* resource. When a font is assigned to a *font_table* resource, at run-time Mrm automatically creates an XmFontList that contains only the specified font. A font value can also be used as an element in font_table, although in this context it must be private to the UIL module.

The font type can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal.

In some versions of UIL, the default character_set is always ISO8859-1, instead of being based on the LANG environment variable or XmFALLBACK_CHARSET.

The UIL compiler may exit with a severe internal error if a user-defined *character_set* is used in a font that is exported or specified as a resource value. If this problem occurs in your version of UIL, only predefined *character_set* values can be used in *font*, *fontset*, and *font_table* values. The workaround is to specify these problematic values in an X resource file.

Example

```
...
procedure
      change_font (font);

value
      title_font          : font ('-*-helvetica-bold-r-nor-
      mal-*-160-100-100-*-iso8859-1');
      family              : 'courier';
      style               : 'medium';
      body_font           : font ('-*-' & family &'-'& style & '-r-nor-
      mal-*-120-100-100*-iso8859-1');
      kanjiFont           : font ('-*-JISX0208.1983-1', character_set = jis_kanji);
      default_font        : imported font;

value
      XtNheadlineFont : argument (‘headlineFont’, font);

object label: XmLabel {
      arguments {
            XmNfontList = title_font;
      };
};
...
```

See Also

`character_set`(6), `fontset`(6), `font_table`(6).

Name

font_table – Motif font list type.

Synopsis

Syntax:

font_table ([*character_set* =] *font_expression* [, ...])

MrmType:

MrmRtypeFontList

Description

A *font_table* value represents a Motif XmFontList. An XmFontList is a data type that specifies the fonts that are in use. Each entry in a font list specifies a font or a font set and an associated tag. When a Motif compound string (XmString) is displayed, the font list tag for the string is used to match the string with a font or a font set, so that the compound string is displayed appropriately.

In UIL, a font_table is represented literally by the symbol *font_table*, followed by a list of one or more *font* or *fontset* values. The elements of a *font_table* must be defined as private values. The character_set of an entry in the list can be overridden by preceding it with a predefined or user-defined *character_set* and an equal sign (=).

Usage

The *font_table* type can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an *argument* literal. A *font_table* is converted to an XmFontList at run-time by Mrm.

The UIL compiler may exit with a severe internal error if a user-defined character_set is used in a *font_table* that is exported or specified as a resource value. This situation can occur if *character_set* is specified directly or indirectly in one of the entries. If this problem occurs in your version of UIL, only predefined *character_set* values can be used in *font*, *fontset*, and *font_table* values. The workaround is to specify these problematic values in an X resource file.

Example

```
...
procedure
    switch_styles (font_table);

value
    latin1    : font ('*-iso8859-1', character_set = iso_latin1);
    hebrew    : font ('*-iso8859-8', character_set = iso_hebrew);
    list      : font_table (latin1, hebrew);
```

```
value
      XtNdefaultFonts : argument ('defaultFonts', font_table);

object label: XmLabel {
      arguments {
            XmNfontList = list;
      };
};
...
```

See Also

`XmFontListAppendEntry`(1), `XmFontListEntryCreate`(1), `XmFontListEntryLoad`(1), `character_set`(6), `font`(6), `fontset`(6).

Name

fontset – XFontSet type.

Synopsis

Syntax:

fontset (*string_expression* [, ...] [, character_set = *character_set*])

MrmType:

MrmRtypeFontSet

Description

A *fontset* value represents an XFontSet, which is an Xlib structure that specifies all of the fonts that are needed to display text in a particular locale. A fontset is represented literally by the symbol *fontset*, followed by a list of string expressions that evaluate to font names and an optional *character_set*. All parts of the string expressions that make up the list of font names must be private to the UIL module. The *character_set* is associated with the fontset if it appears in a *font_table*. If *character_set* is not specified, it is determined from the codeset portion of the LANG environment variable if it is set, or XmFALLBACK_CHARSET otherwise.

The string expression that specifies the font name is a list or wildcarded set of X Logical Font Description (XLFD) strings. This list is stored in the UID file and used as a parameter to XCreateFontSet() at run-time to load the font set. See Volume One, *Xlib Programming Manual*, and Volume Two, *Xlib Reference Manual*, for more information on fonts.

Usage

You can use a *fontset* value to specify a *fontset* or *font_table* resource. When a *fontset* is assigned to a *font_table* resource, at run-time Mrm automatically creates an XmFontList that contains only the specified fontset. A fontset value can also be used as an element in font_table, although in this context it must be private to the UIL module.

The *fontset* type can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an *argument* literal.

In some versions of UIL, the default character set is always ISO8859-1, instead of being based on the LANG environment variable or XmFALLBACK_CHARSET.

The UIL compiler may exit with a severe internal error if a user-defined *character_set* is used in a fontset that is exported or specified as a resource value. If this problem occurs in your version of UIL, only predefined *character_set* values can be used in *font*, *fontset*, and *font_table* values. The workaround is to specify these problematic values in an X resource file.

Example

```
procedure
    change_fontset (fontset);

value
    japanese_font : fontset (‘-misc-fixed-*-75-75-*’);
    default_font : imported font;

value
    XtNbodyFontSet : argument (‘bodyFontSet’, fontset);

object label: XmLabel {
    arguments {
        XmNfontList = japanese_font;
    };
};
```

See Also

`character_set`(6), `font`(6), `font_table`(6).

Name

icon – multi-color rectangular pixmap type.

Synopsis

Syntax:

icon ([color_table = *color_table_name* ,] *row* [, ...])

MrmType:

MrmRtypeIconImage

Description

An *icon* value represents a multi-color rectangular pixmap, or array of pixel values. An icon is represented literally by the symbol *icon*, followed by an optional *color_table* specification and a list of strings that represent the rows of pixel values in the *icon*.

If a *color_table* is specified, it must be a private value and cannot be forward referenced. If a *color_table* is not specified, the following default color_table is used:

color_table (background_color = ' ', foreground color = '*')

Each *row* in the *icon* is a character expression that represents a row of pixel values. Each character in the *row* represents a single pixel. All of the rows in the *icon* must be the same length and must contain only characters defined in the *color_table* for the *icon*. The UIL compiler generates an error if these rules are violated.

Usage

The type *icon* can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal. An *icon* can be retrieved by an application with `MrmFetchIconLiteral`() or `MrmFetchBitmapLiteral`().

When an *icon* is specified as a resource value for a widget, the depth of the pixmap created by Mrm at run-time is the same as the depth of the widget. When an *icon* is retrieved with `MrmFetchIconLiteral`(), the depth of the resulting pixmap is the value returned from the `DefaultDepthOfScreen`() macro. When an *icon* is retrieved with `MrmFetchBitmapLiteral`(), the depth of the resulting pixmap is always one. The *color_table* of an *icon* retrieved with this function must only contain mappings for background color and foreground color or the function fails and returns MrmNOT_FOUND.

The UIL compiler may not check the type of the value specified as the *color_table* for an *icon*. If the compiler allows the specification of a value that is not a *color_table*, it generates an error message when the *icon* is referenced. If no reference to the *icon* occurs in the module, the compiler exits with a severe internal error.

If the *row* values in an *icon* literal do not consist entirely of string literals, the UIL compiler may generate an error message or crash with a segmentation violation.

If a named value is declared as an imported *icon* in one UIL module file, but defined with a different type in another, an error is generated at run time when Mrm attempts to retrieve the *icon*. If you attempt to define a named variable with the value of an *icon* variable, the UIL compiler may generate a large number of errors that are seemingly unrelated to the assignment.

Example

```
...
value
    ! Define an icon that uses default color table and can be retrieved
    ! as a resource or with any of the fetch procedures including
    ! MrmFetchBitmapLiteral():
    checker : icon ('* *',  ' * ',  '* *');
    ! Define an icon that uses a custom color table which contains named
    ! colors. This icon cannot be retrieved with MrmFetchBitmapLiteral().
    red_blue : color_table (color('red') = 'r', color('blue') = 'b');
    plus : icon (color_table = red_blue, 'brb', 'rrr', 'brb');
    ! Declare an argument of type icon.
    XtNwmIcon : argument (‘wmIcon’, icon);

! Declare a procedure taking an icon parameter.
procedure
    display_icon (icon);

! Use an icon for a resource value
object label: XmLabel {
    arguments {
        XmNlabelType = XmPIXMAP;
        XmNlabelPixmap = plus;
    };
};
...
```

See Also

`MrmFetchBitmapLiteral`(3), `MrmFetchIconLiteral`(3), `MrmFetchSetValues`(3), `MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3), `color_table`(6), `pixmap`(6), `xbitmapfile`(6).

Name

integer – whole number type.

Synopsis

Syntax:

[+ | -]0-9[...]

MrmType:

MrmRtypeInteger

Description

An *integer* value represents a negative or positive whole number. An *integer* is represented literally by an optional sign followed by one or more consecutive digits.

An integer can also be represented literally by the symbol *integer* followed by a *float*, *single_float*, or *boolean* expression. The expression is converted to an *integer* and can be used in any context that an *integer* value is valid. An *integer* is formed from a *float* or *single_float* by truncating the fractional value. You can add 0.5 to the *float* or *single_float* value if rounding is desired. If a *float* or *single_float* larger (smaller) than MAXINT (-MAXINT) is converted to an *integer*, the resulting value is MAXINT (MININT). An *integer* is formed from a *boolean* by converting *true* and *on* to 1 and *false* and *off* to 0.

Usage

The allowable range of an *integer* value is determined by the size of an integer on the machine where the UIL module is compiled. Since an integer on most architectures is typically a minimum of four bytes, *integer* values may safely range from -2147483647 (-MAXINT) to 2147483647 (MAXINT). You can ensure greater portability by keeping *integer* values within the four-byte range. The UIL compiler generates an error if it encounters an *integer* outside of the machine's representable range.

The type *integer* can be used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal.

Widget resources of type Position (short) and Dimension (unsigned short) are specified as *integers* in UIL. As a result, the UIL compiler does not generate an error if an out-of-range value is assigned to such a resource. If the sizeof(short) is smaller than sizeof(int), part of the out-of-range value is truncated, which produces an undefined result. The part truncated depends on the C compiler and byte-ordering of the machine on which the UIL module is compiled. For maximum portability, Position values should be limited to the range -32768 to 32767 and Dimension values should be limited to the range 0 to 65536.

The UIL compiler uses -MAXINT to MAXINT, not MININT to MAXINT, as the allowable range for integers, which means that on an architecture with four-byte integers, the minimum *integer* value allowed is -2147483647, not -2147483648. The value MININT can be used, however, by converting a float smaller than -MAXINT to an *integer*.

Example

```
...
! Declare a procedure taking an integer value.
procedure
    set_speed (integer);

! Define some integer variables.
value
    meaning_of_life: 41;
    the_question    : imported integer;
    half_life       : meaning_of_life / 2;
    ten             : integer (10.75);
    round_factor    : 0.5;
    eleven          : integer (10.75 + round_factor);
    one             : integer (true);
    ! Generate MININT value by converting large negative float:
    minint          : integer (-3.0e30);

! Define an argument of type integer.
value
    XtNsize : argument ('size', integer);

object pb : XmPushButton {
    arguments {
        XmNleftOffset = -3;
    };
};
...
```

See Also

`boolean`(6), `float`(6), `integer_table`(6), `single_float`(6).

Name

integer_table – array of integers.

Synopsis

Syntax:

integer_table (*integer_expression* [, ...])

MrmType:

MrmRtypeIntegerVector

Description

An *integer_table*[1] value represents an array of integers. An integer_table is represented literally by the symbol *integer_table*, followed by a list of integer expressions.

Usage

The type name *integer_table* can be used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal. In Motif 1.2, the XmNselectionArray resource of the XmText and XmTextField widgets is the only built-in *integer_table* resource. When the resource is set, UIL automatically sets the XmNselectionArrayCount resource to the number of elements in the array. In Motif 2.0 and later, the XmNselectedPositions resource of the List, and the XmNdetailOrder resource of the Container are also built-in *integer_table* types. The XmNselectedPositionCount and XmNdetailOrderCount resources are automatically set by UIL respectively.

Unlike *asciz_string_table* and *compound_string_table* values, an *integer_table* is not NULL-terminated. As a result, you must either use *integer_table* values of a set length, include the length explicitly, or use a value to indicate the end of the array. The application code that uses the values must use the same conventions as the UIL module.

Example

```
...
value
    ! Define table with known number of elements (12).
    days: integer_table (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
    ! Define table with length as first element.
    grades : integer_table (5, 95, 87, 100, 92, 82);
    ! Define table with last element of MININT.
    end_of_table : integer (3.0e-30);
```

1.Erroneously given as *integer_type* in 1st edition.

```
    ages : integer_table (25, 29, 29, 30, 32, end_of_table);

! Declare a procedure taking an integer_table
procedure
    compute_average (integer_table);

! Declare an argument taking an integer table
value
    XtNdaysPerMonth : argument (‘daysPerMonth’, integer_table);
...
```

See Also

`integer`(6).

Name

keysym – character type.

Synopsis

Syntax:

keysym (*string_literal*)

MrmType:

MrmRtypeKeysym

Description

A *keysym* value is used to represent a single character. A keysym is represented literally by the symbol *keysym*, followed by a string value that contains exactly one character. If the string is a variable, it can be forward referenced and must be private to the UIL module.

Usage

A *keysym* value is typically used to specify a widget mnemonic resource, such as XmNmnemonic. The *keysym* type can be used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal.

When a *keysym* is retrieved by an application with `MrmFetchLiteral`(), the value argument returned is the character value of the *keysym*, not a pointer to the value like many other types.

The UIL compiler may not generate an error if the string expression in a *keysym* literal is more than one character long, but an error will be generated by Mrm at run-time. If an invalid *keysym* is specified as a resource value, the resource is not set. If the application attempts to retrieve an invalid *keysym* with `MrmFetchLiteral`(), MrmNOT_FOUND is returned.

Example

```
...
procedure
    set_keysym (keysym);

value
    d_key : keysym (‘d’);
    XtNquitKey : argument (‘quitKey’, keysym);

object the_button : XmPushButton
    arguments {
        XmNmnemonic = keysym (‘b’);
    };
```

```
};
...
```

See Also

`MrmFetchLiteral`(3).

Name

pixmap – generic icon or xbitmapfile type.

Synopsis

Syntax:

No literal syntax.

MrmType:

MrmRtypeIconImage or MrmRtypeXBitmapFile

Description

A *pixmap* value can be either an *icon* or *xbitmapfile*. In either case, the type specifies an array of pixel values. A *pixmap* does not have its own literal representation; a *pixmap* value is specified with either the *icon* or *xbitmapfile* literal syntax.

Usage

The type *pixmap* can be used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal. The purpose of the *pixmap* type is to allow either an *icon* or an *xbitmapfile* value to be imported, passed as a callback argument, or specified as a resource value.

Example

```
...
value
    ! Declare an imported pixmap that can be defined as an icon or xbitmapfile.
    stop_pixmap : imported pixmap;
    ! Declare an argument to which an icon or xbitmapfile can be assigned.
    XtNstipplePixmap : argument (‘stipplePixmap’, pixmap);

! Declare a procedure to which an icon or xbitmapfile can be passed.
procedure
    print_pixmap (pixmap);
...
```

See Also

`icon`(6), `xbitmapfile`(6).

Name

reason – user-defined callback type.

Synopsis

Syntax:

reason (*string_expression*)

MrmType:

none

Description

A *reason* value represents a user-defined callback. A reason is represented literally by the symbol *reason*, followed by a string expression that evaluates to the name of a callback. The name of the *reason* is assigned to the name member of the ArgList structure passed to `XtSetValues`(). The name is typically the name of a callback with the XmN or XtN prefix removed.

Usage

A user-defined callback can be used in the callbacks section of a UIL module for both built-in Motif widgets and user-defined widgets. While user-defined callbacks are typically assigned to a named variable in the value section, they can also be specified literally in the *arguments* section of an *object* definition. If you are defining arguments for a widget or widget set which is not predefined, you should define them as named variables in a separate UIL module that can be included by any module that uses the widget(s).

Reasons must be private values; they cannot be imported or exported. The UIL compiler allows imported and exported declarations, but it generates an error when the user-defined *reason* is used. Since *reason* values cannot be exported, they cannot be retrieved by an application.

The *reason* type can only be used to define callback resource types. The argument type is used to specify other user-defined resources.

Example

From *Xaw/Panner.uih*:

```
! Resources and definitions for the Athena Panner widget.
...
! Callback definitions
value
    XtNreportCallback = reason ('XtNreportCallback');
    ...
```

From *my_module.uil*:

```
include file ’Xaw/Panner.uih’;

procedure
      panner_report();

object panner : user_defined procedure XawCreatePanner {
      callbacks {
            XtNreportCallback = procedure panner_report();
      };
};
...
```

See Also

`MrmRegisterClass`(3), `include`(5), `object`(5), `argument`(6).

Name

rgb – color specified with the values of red, green, and blue components.

Synopsis

Syntax:

rgb (*red_integer*, *green_integer*, *blue_integer*)

MrmType:

MrmRtypeColor

Description

The type *rgb* represents a color as a mixture of red, green, and blue values. An rgb value is represented literally by the symbol *rgb*, followed by a list of three integers that specify the red, green, and blue components of the color. The amount of each color component can range from 0 (0 percent) to 65,535 (100 percent). Mrm allocates rgb values with `XAllocColor`(). See Volume 1, *Xlib Programming Manual*, and Volume 2, *Xlib Reference Manual*, for more information on color allocation.

Usage

An *rgb* value or literal can be used anywhere a color value is expected: as a callback argument, as a resource value, or in a *color_table*. Unlike color values, it is not possible to specify a foreground or background fallback for *rgb* values. For this reason, and to maximize the number of shareable color cells, you should use named colors defined with the color type whenever possible.

If a color cannot be allocated, Mrm substitutes black, unless the color is specified as the background color or foreground color in a *color_table* and the foreground color or background color is already black. In this situation, white is substituted.

In Motif version 1.2.1, the color substitutions described above do not take place. When a color allocation fails for an rgb value specified directly or indirectly (in the color_table of an icon) the resource is not set. If the allocation fails in a call to `MrmFetchColorLiteral`() or `MrmFetchIconLiteral`(), MrmNOT_FOUND is returned. In Motif 2.1, `XBlackPixelOfScreen`() is used where `XAllocColor`() fails.

Note that the values that specify that red, green, and blue components cannot be integer expressions. The UIL compiler, however, does not generate an error if an integer expression is encountered; it silently replaces the expression with the value 0. In addition, the UIL compiler does not report an error if an integer specified for a color value is less than 0 or greater than 65,535. If any of the three components is out-of-range, the three values stored in the UID file are undefined.

Example

```
value
    white  : rgb (65535, 65535, 65535);
    orange : exported rgb (65535, 32767, 0);
    grape  : imported rgb;
    ctable : color_table (white = ’w, orange = ’o’, grape = ’g’);
    ....

object label : XmLabel {
    arguments {
        XmNforeground = rgb (0, 0, 32767);
        XmNbackground = orange;
    };
};
```

See Also

`MrmFetchColorLiteral`(3), `MrmFetchIconLiteral`(3), `MrmFetchSetValues`(3), `MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3),
`color`(6), `color_table`(6), `icon`(6).

Name

single_float – single-precision floating point type.

Synopsis

Syntax:

single_float (*numeric_expression*)

MrmType:

MrmRtypeSingleFloat

Description

A *single_float* value represents a negative or positive single-precision floating point number. A single_float is represented literally by the symbol *single_float*, followed by a boolean, float or integer expression. The expression is converted to a *single_float* and can be used in any context in which a *single_float* value is valid. A *single_float* is formed from a boolean by converting true and on to 1.0 and false and off to 0.0. If a float expression is greater than (less than) the largest (smallest) representable float, the resulting *single_float* is +infinity (-infinity).

Usage

The type *single_float* can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal. A *single_float* value is used to save space, as the storage used by a *single_float* is usually less than that used by a float.

The allowable range of a *single_float* value is determined by the size of a C float on the machine where the UIL module is compiled. Since a float on most architectures is typically a minimum of four bytes, *single_float* values may safely range from 1.4013e-45 to 3.40282e+38 (positive or negative). You can ensure greater portability by keeping *single_float* values within the four-byte range.

Example

```
...
! Declare a procedure taking a single_float value.
procedure
    sqrt (single_float);

value
    avogadro   : single_float (6.023e+23);
    prime_rate : imported single_float;

! Define an argument of type single_float.
value
    XtNarea : argument ('area', single_float);
...
```

See Also

`boolean`(6), `float`(6), `integer`(6).

Name

string – NULL-terminated character string type.

Synopsis

Syntax:

[#*character_set*] "*character_expression"* or
’*character_expression’*

MrmType:

MrmRtypeChar8

Description

A *string* value represents a NULL-terminated single-byte, multi-byte, or wide-character string. A *string* literal is represented by either a double or single-quoted sequence of characters, that may be up to 2000 characters long. Newer versions of UIL may allow even longer strings. The type of quotes used to delimit a *string* literal determines how the string is parsed by the UIL compiler.

Both double and single-quoted strings can directly contain characters with decimal values in the range 32 to 126 and 160 to 255. Characters with values outside of the range can only be entered using the escape sequence \value\, where value represents the character code desired. To allow the easy specification of commonly-used non-printing characters codes, UIL recognizes the following escape sequences:

Character	**Meaning**
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\’	Single quote
\”	Double quote

A double-quoted string consists of an optional *character_set*, followed by a sequence of characters surrounded by a double quotes. Double-quoted strings cannot span multiple lines, but may contain the \n escape sequence. If a *character_set* is specified, it precedes the string and is indicated by a pound sign (#). Either a built-in or user-defined *character_set* can be specified. If a *character_set* is not specified, the default character set of the module is used. The default *character_set* can be specified with the *character_set* option in the module header of a UIL module. If this option is not set, the default is determined from the codeset portion of the LANG environment variable if it is set, or XmFALLBACK_CHARSET otherwise.

If the UIL compiler is invoked with the -s option, double-quoted strings are parsed in the current locale. When UIL parses localized strings, escape sequences may be interpreted literally. You can avoid unexpected results by restricting the use of escape sequences to single quoted strings.

A single-quoted string consists of a sequence of characters surrounded by single quotes. Unlike double-quoted strings, single-quoted strings can span multiple lines by using a backslash (\) to indicate that the string is continued on the next line. The newline character following the backslash is not included in the string. The \n escape sequence should be used if an embedded newline is desired. The *character_set* of a single-quoted string defaults to the codeset portion of the LANG environment variable if it is set, or XmFALLBACK_CHARSET otherwise.

The parsing direction of either string variant is determined by the *character_set* of the string. A string that is parsed right-to-left is stored in the UID file in the reverse order that it appeared in the UIL source module. The parsing direction and *character_set* writing direction determine the order of individual characters when a string is printed or displayed. The writing direction of a string is generally the same as the parsing direction, unless explicitly overridden in a compound_string literal. The order of the characters in escape sequences is always the same, regardless of the parsing direction.

Usage

A single or double-quoted string value can be used anywhere a string or string expression is expected. A string expression can be a single string value or two or more string values concatenated with the string concatentation operator (&). A string or string expression can also be used anywhere a *compound_string* is expected, since the UIL compiler automatically converts the string to a compound string, with the character set determined by the rules described above. (When determining the character set, the UIL compiler may use ISO8859-1 as the fallback character set, even if the value has been changed by the vendor. Therefore, you should specify a character set explicitly instead of relying on XmFALLBACK_CHARSET.)

Any newline characters in a NULL-terminated string that is converted into a *compound_string* are not converted into separator components to make a multi-line compound string. If you need a multi-line compound string, it must be specified as a concatenated set of values using the compound_string literal syntax with the separate property set to true.

The type *string* can be used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal. String values used in string expressions or in *compound_string* literals must be private to the module in which they are used.

Example

```
...
procedure
    tie_knot (string);

value
    display         : imported string;
    skit_name       : 'Unfrozen Caveman Lawyer';
    hello           : #iso_hebrew"\237\229\236\249\";
    quote           : exported 'Quote the Raven, 'Nevermore.\'\n';
    concat          : 'The Cat' & ' in the Hat';
    multi           : 'All that we see or seem\nIs but a dream within a dream.';
    ! Define a resource of type string.
    XtNfilename     : argument ('filename', string);

object play : XmPushButton {
    arguments {
        ! String automatically converted to XmString
        XmNlabelString = skit_name;
    };
};
...
```

See Also

`asciz_string_table`(6), `character_set`(6), `compound_string`(6), `compound_string_table`(6).

Name

translation_table – Xt translation table type.

Synopsis

Syntax:

translation_table ([‘#override‘ | ’#augment’ | ’#replace’] *string_expression* [, ...])

MrmType:

MrmRtypeTransTable

Description

A *translation_table* value represents an X Toolkit translation table. A translation table is a list of translations, where each translation maps an event or an event sequence to an action name. In UIL, a translation_table is represented literally by the symbol *translation_table*, followed by an optional directive and list of string expressions that are interpreted as translations. If specified, the directive must be one of #override, #augment, or #replace. The translations are specified as a list of string expressions, one per translation. The individual translations are concatenated and separated with newline characters before they are stored in the UID file.

Usage

The *translation_table* type can be used as the type of an imported value, as a parameter type in a *procedure* declaration, or as the type in an *argument* literal.

The syntax of a *translation_table* is not verified by the UIL compiler. Instead, Mrm converts a translation_table literal to an XtTranslations value with `XtParseTranslationTable`() at run-time. Errors that occur when parsing the *translation_table* are passed to `XtWarning`(). Because `XtParseTranslationTable`() always returns a valid XtTranslations value, even when parsing errors occur, the run-time conversion of a *translation_table* cannot fail. See Volume 4, *X Toolkit Intrinsics Programming Manual*, and Volume 5, *X Toolkit Intrinsics Reference Manual*, for more information about translation tables.

Example

```
...
procedure
    set_translations (translation_table);
    exit();

value
    XtNquickKeys : argument (’translations’, translation_table);

value
```

```
        quit_tt : translation_table ('#override', '<Key>q: ArmAndActivate()');
        other_tt : imported translation_table;

object quit : XmPushButton {
        arguments {
                XmNtranslations = quit_tt;
        };
        callbacks {
                XmNactivateCallback = procedure exit();
        };
};
...
```

See Also

`MrmFetchLiteral`(3).

Name

wide_character – wide-character string type.

Synopsis

Syntax:

wide_character (*string_expression*)

MrmType:

MrmRtypeWideCharacter

Description

A *wide_character* value represents a wide-character string. The corresponding C type is wchar_t *. A wide_character literal is represented by the symbol *wide_character*, followed by a string expression.

Usage

A *wide_character* literal is used to make the UIL compiler parse a regular character string as a wide-character string. A *wide_character* string is parsed with the `mbstowcs`() function. The operation of this function depends on the setting of the locale. See the uil reference page for more information regarding the locale setting. The *wide_character* literal syntax may not work in early releases of Motif 1.2. However, you can specify a wide-character string using the normal UIL string syntax. The difference is that the UIL compiler does not verify that a wide-character string specified in this way is properly formed.

The type *wide_character* can also be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal.

Example

```
...
procedure
    print_wcs (wide_character);

value
    wcs : wide_character ('\204\176\224\189\');
    name : imported wide_character;
    XtNwideCharacterString : argument ('wideCharacterString',
    wide_character);

object text : XmText {
    arguments {
        XmNvalueWcs = wcs;
    };
};
...
```

See Also

`uil`(4), `procedure`(5), `argument`(6), `string`(6).

Name

widget – widget type.

Synopsis

Syntax:

See the object section of the UIL file format reference page.

MrmType:

none

Description

Objects that are declared or defined in a UIL object section are of type *widget*. Values of type *widget* are the only UIL values that are not declared or defined in a value section. The literal representation of a *widget* is described in the object section of the UIL file format reference page.

Usage

The type *widget* can be used as a parameter type in a *procedure* declaration or as the type in an *argument* literal. When a widget is used as a callback parameter or resource value in the declaration of another widget, it must be part of the same hierarchy as that widget. A widget hierarchy is defined by the widget passed to `MrmFetchWidget`() or `MrmFetchWidgetOverride`() and it includes all of the descendants of that widget. If you need to specify a widget in a different hierarchy as a callback parameter, you can use the string name of the widget instead and convert it to a widget pointer in the callback with `XtNameToWidget`().

Widgets can be forward referenced. If Mrm encounters a reference to a widget that has not been created in the current hierarchy, it creates the remainder of the hierarchy and makes another attempt to resolve the reference. If the reference cannot be resolved at that point, Mrm does not add the callback or set the resource for which the widget is specified. As of Motif 1.2, Mrm does not generate a warning when a widget reference cannot be resolved.

Prior to Motif 1.2.1, the UIL compiler generates an error when widget is used as a *procedure* parameter or type in an *argument* literal. To work around this problem, you can use the type *any*.

Example

```
...
value
    ! Declare Athena tree widget constraint argument.
    XtNtreeParent : argument (’treeParent’, widget);
```

```
procedure
    manage (widget);

object
    button1 : XmPushButton {
        callbacks {
            XmNactivateCallback = manage (button3);
        }
        arguments {
            XmNbottomAttachment = XmATTACH_FORM;
            XmNbottomOffset = 40;
            XmNrightAttachment = XmATTACH_WIDGET;
            XmNrightWidget = button1;
        };
    };
    button2 : XmPushButton { };
    button3 : XmPushButton {
        arguments {
            XmNbottomAttachment = XmATTACH_FORM;
        };
    };
    form : XmForm {
        controls {
            XmPushButton button1;
            XmPushButton button2;
            unmanaged XmPushButton button3;
        };
    };
...
```

See Also

`MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3), `object`(5), `procedure`(5), `argument`(6).

Name

xbitmapfile – X bitmap file type.

Synopsis

Syntax:

xbitmapfile (*string_expression*)

MrmType:

MrmRtypeXBitmapFile

Description

An *xbitmapfile* value represents a file that contains a bitmap in the standard X bitmap file format. An xbitmapfile literal is represented by the symbol *xbitmap-file*, followed by a string expression that evaluates to the name of the file containing the bitmap. The X bitmap is loaded at run-time by Mrm using `XmGetPixmapByDepth`(). See Volume 1, *Xlib Programming Manual*, for more information about the X bitmap file format.

Usage

The type *xbitmapfile* can be used as the type of an imported value, as a parameter type in a procedure declaration, or as the type in an argument literal. An *xbitmap-file* value can be retrieved by an application with `MrmFetchIconLiteral`(). The `MrmFetchBitmapLiteral`() procedure cannot be used to retrieve values of the *xbitmapfile* type.

When an *xbitmapfile* is specified as a resource value for a widget, the depth of the pixmap created by Mrm at run-time is the same as the depth of the widget. When an *xbitmapfile* is retrieved with `MrmFetchIconLiteral`(), the depth of the resulting pixmap is the value returned from the `DefaultDepthOfScreen`() macro.

The UIL compiler stores the specified file name in the UID output file, not the X bitmap to which the name refers. The compiler does not verify that the specified file exists. If an *xbitmapfile* specified as a resource cannot be loaded, the resource is not set. If MrmFetchIconLiteral fails to load an *xbitmapfile*, MrmNOT_FOUND is returned.

Example

```
...
! Declare a bitmap of the most challenging ski slope in the Northeast.
value
    goat: xbitmapfile ('goat.xbm');

object scary : XmLabel {
    arguments {
```

```
                XmNlabelType = XmPIXMAP;
                XmNlabelPixmap = goat;
        };
};
...
```

See Also

`XmGetPixmapByDepth`(2), `MrmFetchBitmapLiteral`(3), `MrmFetchIconLiteral`(3), `MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3), `icon`(6), `pixmap`(6).