User-defined Locales [A]

In order to provide more flexible support for multicultural software interfaces, the UNICOS system supports the concept of a locale. A *locale* is a collection of culture-dependent information used by an application to interact with a user. The information in a locale includes information on the following:

- Sorting or collation (`LC_COLLATE`)
- Character classes and case mapping (`LC_CTYPE`)
- Basic interaction messages (`LC_MESSAGES`)
- Monetary formats (`LC_MONETARY`)
- Numeric formats (`LC_NUMERIC`)
- Time and date formats (`LC_TIME`)

To make use of these features in an application, that application must be written to use the information in a locale or interfaces that access the locale implicitly. Use of this information from a locale can help an application be free of cultural dependencies. Such an application is said to be *internationalized*. The goal is that such an application can then be run by a user and, through the manipulation of environment variables, interact with that user in a more natural manner.

A.1 The `localedef` utility

Locales are defined using the `localedef` utility. The input to `localedef` is a text file (known as a *locale definition file*) that describes all the attributes of the desired locale. Using this information, `localedef` creates files that can be loaded by application (using the `setlocale`(3) library routine) to establish that locale in the environment of the application.

The `localedef` utility is invoked in the following manner:

```
localedef [ -c ] [ -i localefile ] [ -f charmap ] locale
```

The optional arguments are used as follows:

`-c` Creates the locale even if warning messages are issued. By default, the locale will not be created if any warnings occur.

`-i` *localedef*	Specifies the name of the locale definition file. If not specified, the locale definition will be read from the standard input.
`-f` *charmap*	Indicates the character encoding to be used. Currently two are supported: 646 (which supports the ISO 646 or ASCII character encoding) and 8859 (which supports the ISO 8859-1 character encoding). The default is 646.

The specified locale is the name of the locale to be created. If *locale* contains a slash character, it is interpreted as a path name of a directory to put the locale files in (if the directory does not exist, it will be created). Otherwise, the locale will be created in `/usr/lib/locale`, making it generally available for users.

A.1.1 Character specifications

In a locale definition file, characters can be specified symbolically or as literal values. The use of symbolic values is preferred, because this allows the locale definition file to be independent of the particular character encoding.

Literal values can either be the specific character itself (assuming that the target locale will have the same encoding for that character as the 646 locale) or a numeric value. Numeric values can be of the following forms (assuming that the backslash (\) is the current escape character):

`\xNN` *For hexadecimal byte values*
`\dNNN` *For decimal byte values*
`\NNN` *For octal byte values*

A multibyte numeric value can be specified by concatenating byte specifications of the above form.

Characters can also be specified symbolically in a locale definition file. For example, the encoding of the characters 'a', '5', or the bell character can be specified as < `a` >, < `five` >, and < `alert` >. The symbolic names for characters in the 646 and 8859 charmap are specified in the following list:

Name	Value
`<NUL>`	\x00
`<SOH>`	\x01
`<STX>`	\x02
`<ETX>`	\x03

<EOT>	\x04
<ENQ>	\x05
<ACK>	\x06
<BEL>	\x07
<alert>	\x07
<backspace>	\x08
<tab>	\x09
<newline>	\x0a
<vertical-tab>	\x0b
<form-feed>	\x0c
<carriage-return>	\x0d
<SO>	\x0e
<SI>	\x0f
<DLE>	\x10
<DC1>	\x11
<DC2>	\x12
<DC3>	\x13
<DC4>	\x14
<NAK>	\x15
<SYN>	\x16
<ETB>	\x17
<CAN>	\x18
<EM>	\x19
<SUB>	\x1a
<ESC>	\x1b
<IS4>	\x1c
<IS3>	\x1d
<IS2>	\x1e
<IS1>	\x1f
<SP>	\x20
<space>	\x20

<exclamation-mark>	!
<quotation-mark>	"
<number-sign>	#
<dollar-sign>	$
<percent-sign>	%
<ampersand>	&
<apostrophe>	'
<left-parenthesis>	(
<right-parenthesis>	)
<asterisk>	*
<plus-sign>	+
<comma>	,
<hyphen>	-
<hyphen-minus>	-
<period>	.
<full-stop>	.
<slash>	/
<solidus>	/
<0> or <zero>	0
<1> or <one>	1
<2> or <two>	2
<3> or <three>	3
<4> or <four>	4
<5> or <five>	5
<6> or <six>	6
<7> or <seven>	7
<8> or <eight>	8
<9> or <nine>	9
<colon>	:
<semicolon>	;
<less-than-sign>	<

<equals-sign>	=
<greater-than-sign>	>
<question-mark>	?
<commercial-at>	@
<A>...<Z>	A...Z
<left-square-bracket>	[
<backslash>	\
<reverse-solidus>	\
<right-square-bracket>	]
<circumflex>	^
<circumflex-accent>	^
<underscore>	_
<low-line>	_
<grave-accent>	‘
<a>...<z>	a...z
<left-brace>	{
<left-curly-bracket>	{
<vertical-line>	|
<right-brace>	}
<right-curly-bracket>	}
<tilde>	~
<DEL>	\x7f
<PAD>	\x80
<HOP>	\x81
<BPH>	\x82
<NBH>	\x83
<IND>	\x84
<NEL>	\x85
<SSA>	\x86
<ESA>	\x87
<HTS>	\x88

<HTJ>	\x89
<VTS>	\x8a
<PLD>	\x8b
<PLU>	\x8c
<RI>	\x8d
<SS2>	\x8e
<SS3>	\x8f
<DCS>	\x90
<PU1>	\x91
<PU2>	\x92
<STS>	\x93
<CCH>	\x94
<MW>	\x95
<SPS>	\x96
<EPA>	\x97
<SOS>	\x98
<SGCI>	\x99
<SCI>	\x9a
<CSI>	\x9b
<ST>	\x9c
<OSC>	\x9d
<PM>	\x9e
<APC>	\x9f
<nobreakspace>	\xa0

A.1.2 General syntax of the locale definition file

The format of the locale definition file is a list of category specifications. Each category corresponds to the basic groups of locale information: `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, and `LC_TIME`. The general format for these categories is the following:

```
category_name
keyword      value
keyword      value...
```

```
END category_name
```

`category_name` is either `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, or `LC_TIME`. The specific keywords and the valid associated values are detailed below. The possible values can be strings (enclosed in quotes) or integers or lists of either of these. If the value is a list, then the list elements are separated by semicolons.

Note that the format of the remaining categories, `LC_COLLATE` and `LC_CTYPE`, is quite different and is unique for each of these categories. The details of all the category specifications is described in the following sections.

In addition to the list of category specifications, the locale definition file can have the following global statements:

```
escape_char     value
comment_char    value
```

These define the character used to precede comments, and escape the usual meaning of a character. The default values for these are the following:

```
escape_char
comment_char    #
```

The comment character must appear as the first character of a line. It causes `localedef` to ignore the rest of that line. The escape character is used to specify numeric character constants and to do line continuation. The latter is necessary since each `localedef` directive must appear on a single line. The escape character allows the breaking up of long lines while allowing `localedef` to consider such a set of lines as a single line.

The following example shows specification of the `LC_MONETARY` category:

```
LC_MONETARY
int_curr_symbol      "<U><S><D><space>"
currency_symbol      "<dollar-sign>"
mon_decimal_point    "<period>"
mon_thousands_sep    "<comma>"
mon_grouping         3
positive_sign        "<plus-sign>"
negative_sign        "<hyphen-minus>"
int_frac_digits      2
frac_digits          2
p_cs_precedes        1
p_sep_by_space       0
n_cs_precedes        1
```

```
n_sep_by_space       0
p_sign_posn          4
n_sign_posn          4
END LC_MONETARY
```

A.1.3 The LC_MONETARY category

The LC_MONETARY category describes monetary formatting conventions. The following keywords are recognized by localedef in the category:

int_curr_symbol type: string

The international currency symbol. The value is the three character international currency symbol defined by the ISO 4217:1987 standard followed by a character, such as a space, to separate the currency symbol from the value.

currency_symbol type: string

The local currency symbol.

mon_decimal_point type: string

The symbol used as a decimal point for monetary values.

mon_thousands_sep type: string

The symbol used to separate groups of digits for monetary values.

mon_grouping type: list of integers

Describes how mon_thousands_sep is used to separate digit groups. For the nonfractional part of a monetary value, the digits are separated by mon_thousands_sep into groups of the sizes specified in this list beginning from the least significant digits. All groups should be greater than zero other than the last value which may be -1. If the last value is -1, no further grouping of digits will be done; otherwise, the last grouping value will be used to determine the size of all subsequent groups. As an example, if the value was 1;2;-1 then the value 123456789 would be formatted as 123456,78,9. A typical use would be to separate thousands of digits for an entire value regardless of length. A value of 3 would produce the desired result in this case.

`positive_sign` type: string

The symbol used to indicate positive monetary values.

`negative_sign` type: string

The symbol used to indicate negative monetary values.

`int_frac_digits` type: integer

The number of fractional digits printed for values formatted with an international currency symbol.

`frac_digits` type: integer

The number of fractional digits printed for values formatted with a local currency symbol.

`p_cs_precedes` type: integer

This value (indicated in parentheses) indicates whether the international and local currency symbols precede (1) or succeed (0) a positive monetary value.

`p_sep_by_space` type: integer

This value indicates that no space separates the international or local currency symbol from a positive monetary value (0), or if a space separates the symbol from the value (1), or if a space separates the symbol and the sign string if adjacent (2).

`n_cs_precedes` type: integer

This value indicates if the international and local currency symbol precedes the value for a negative monetary value (1) or if the symbol succeeds the value (0).

`n_sep_by_space` type: integer

This value indicates that no space separates the international or local currency symbol from a negative monetary value (0), or if a space separates the symbol from the value (1), or if a space separates the symbol and the sign string if adjacent (2).

`p_sign_posn` type: integer

This value indicates the relative position of the positive sign and a positive monetary value.

Value	Description
0	Parentheses enclose the value and the currency symbol (local or international).
1	The sign precedes the value and the currency symbol.
2	The sign succeeds the value and currency symbol.
3	The sign precedes the currency symbol.
4	The sign succeeds the currency symbol.

`n_sign_posn` type: integer

This value indicates the relative position of the negative sign and a negative monetary quantity. The values are the same as for `p_sign_posn` above.

`copy` type: string

Causes the copying of the `LC_MONETARY` specification from the locale specified as the keyword value. This keyword cannot be combined with any of the other keywords in the category.

This category affects the operation of the `strfmon` () library routine. This information is also available directly from the `localeconv` () library routine.

A.1.4 The LC_MESSAGES category

The `LC_MESSAGES` category describes messages for user interaction. Currently this is limited to the format of simple acknowledgment (yes or no) requests. The following keywords are recognized by `localedef` in the category:

`yesexpr` type: string

An extended regular expression defining the possible value for an affirmative response.

`noexpr` type: string

An extended regular expression defining the possible value for a negative response.

`yesstr` type: string

A string defining an affirmative response.

`nostr` type: string

A string defining a negative response.

`copy` type: string

Causes the copying of the `LC_MESSAGES` specification from the locale specified as the keyword value. This keyword cannot be combined with any of the other keywords in the category.

This information is available directly from the `nl_langinfo` () library routine.

A.1.5 The `LC_NUMERIC` category

The `LC_NUMERIC` category describes numeric formatting conventions. The following keywords are recognized by `localedef` in the category:

`decimal_point` type: string

The symbol used as a decimal point for numeric values.

`thousands_sep` type: string

The symbol used to separate groups of digits for numeric values.

`grouping` type: list of integers

Describes how `thousands_sep` is used to separate digit groups.

`copy` type: string

Causes the copying of the `LC_NUMERIC` specification from the locale specified as the keyword value. This keyword cannot be combined with any of the other keywords in the category.

This category affects the operation of the `printf` () and `scanf` () family of library routines. This information is also available directly from the `localeconv` () and `nl_langinfo` () library routines.

A.1.6 The `LC_TIME` category

The `LC_TIME` category describes time and date formatting conventions. The following keywords are recognized by `localedef` in the category:

`day` type: list of strings

A list, which must have seven entries, of the days of the week. Example: `"Monday";"Tuesday";...` (most of the following examples will not use symbolic format for string, for example, `"<M><o><n><d><a><y>"`, for clarity even though this is, strictly, bad form)

`abday` type: list of strings

A list, which must have seven entries, of the abbreviated names of the days of the week. Example: `"Mon";"Tue";...`

`mon` type: list of strings

A list, which must have twelve entries, of the months of the year. Example: `"January";"February";...`

`abmon` type: list of strings

A list, which must have twelve entries consisting of the abbreviated names of the months of the year. Example: `"Jan";"Feb";...`

`d_t_fmt` type: string

The format of a date and time specification. See the description of the `strftime`(3) library interface for the syntax of time/date format strings.

`d_fmt` type: string

The format of a date specification. See the description of the `strftime`(3) library interface for the syntax of time/date format strings.

`t_fmt` type: string

The format of a time specification. See a description of the `strftime` () library interface for the syntax of time/date format strings.

`am_pm` type: list of strings

A list, which must have two entries, of the names of antemeridian and postmeridian periods of the day. Example: `"AM";"PM"`

`t_fmt_ampm` type: string

The form of a date and time specification using a 12-hour clock qualified by the appropriate entry from the `am_pm` list. See a description of the `strftime` () library interface for the syntax of time/date format strings.

`era` type: list of strings

Defines how years are counted and displayed for each era in a locale. Each element of the list indicates how a specific time range will be displayed. Each list entry identifies the range of dates that it corresponds to and the format that should be applied for that era. More specifically, each entry has the following form:

direction`:`*offset*`:`*start_date*`:`*end_date*`:`*era_name*`:`*era_format*

The details of each of these fields is detailed below. Using a simple example, the following describes the BC/AD era convention:

```
"+:1:-0001/12/31:-*:BC:%Ey %EC" ; "+:0:0000/01/01:+*:AD:%EC %Ey"
```

This example will be used to clarify the meaning of the individual fields in an era description.

Field	Description
`direction`	Either a + or - to indicate whether the year of the `start_date` has lower or higher numeric values than the year of the `end_date`.
`offset`	The number of the year closest to `start_date`.
`start_date`	The year, month, and day of the beginning of the era. The year, month, and day should be specified in the format *yyyy/mm/dd*, respectively.
`end_date`	The ending date of the era. This can either be specified in the same format as `start_date` or as either -* (beginning-of-time) or +* (end-of-time).
`era_name`	The name of the era. In the above example, either BC or AD.
`era_format`	The format for the printing days in the era.
`era_d_fmt` type: string	The format of the date in alternative era notation.
`era_t_fmt` type: string	The format of the time in alternative era notation.
`era_d_t_fmt` type: string	The format of the date and time in alternative era notation.

`alt_digits` type: list of strings

The alternate names for digits in a date specification. Example:

```
"1st";"2nd";"3rd";"4th";"5th";"6th" ...
```

Up to 100 alternate names can be specified.

`copy` type: string

Causes the copying of the `LC_TIME` specification from the locale specified as the keyword value. This keyword cannot be combined with any of the other keywords in the category.

This category affects the operation of the `strftime` () and `strptime` () library routines. This information is also available directly from the `nl_langinfo` () library routine.

A.1.7 The LC_CTYPE category

The `LC_CTYPE` category can be used to:

- Define membership of character classes
- Specify case conversion

The members of character classes (such as `alpha`, `digit`, `xdigit` `punct`, `space`) can be defined as in the following example. This specifies that the `alpha` class contains a-z and A-Z.

```
alpha   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
        <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>
```

Alternatively, range specifications can be used. The following is equivalent to the previous example:

```
alpha   <a>;...;<z>;<A>;...;<Z>
```

The possible character classes are the following:

```
alpha   print     phonogram
blank   punct     ideogram
cntrl   space     english
digit   xdigit    number
graph   special
```

Additionally, user-defined character classes can be created. For example, the following defines a character class named `xalpha` that includes all alphabetic characters that are used as hexadecimal digits:

```
charclass      xalpha
xalpha        <a>;...;<f>;<A>;...;<F>
```

It is necessary to declare all user-defined character classes with the *charclass* keyword before the members of that class are specified.

Character class mapping may also be defined via the `toupper` and `tolower` keywords. The following example illustrates this:

```
toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);\
        (<z>,<Z>)
```

Unfortunately, the current version of `localedef` does not support ranges for case mapping, so all of the mapping pairs must be specified explicitly.

A.1.7.1 Character class and case mappings

There are implicit rules and restrictions for building character classes and case mappings, so that all relationships do not need to be specified explicitly.

Membership in a class can implicitly add a character to other classes as well.

- Members of the `upper` or `lower` classes are added to the `alpha` class.
- Members of the `alpha` class are added to the `alnum` class.
- Members of the `digit` classes are added to the `xdigit` and `alnum` classes.
- Members of the `blank` class are added to the `space` class.

- Members of the `alpha`, `digit`, `xdigit`, and `punct` classes are added to the `graph` and `print` classes.

Restrictions on character class membership:

- Members of the `digit` class cannot be in the `upper` or `lower` classes.
- Members of the `alpha` and `xdigit` classes cannot be members of the `space`, `cntrl`, or `punct` classes.
- Members of the `space` character class cannot be members of the `graph` class.
- Members of the `graph` or print classes cannot be members of the `cntrl` class.

Note that the relationships need to be considered in conjunction. For example, since members of the `xdigit` class cannot be members or the `cntrl` class, then neither can members of the `digit` class, since membership in the `digit` class implies members in the `xdigit` class. This is a bit complicated but should not be a problem in defining actual locales, because these relationships simply enforce the logical relationships between classes.

For case conversion, the following actions and restrictions are imposed:

- Each member of the conversion must be members of the upper or lower classes, as appropriate.
- If no conversions are specified, the traditional a-z to A-Z conversion will be done.
- If a `toupper` conversion is specified without a `tolower` conversion, then the `tolower` conversion will be the inverse of the `toupper` conversion.

Note that the a-z to A-Z conversions are not included implicitly in a conversion if that conversion is explicitly defined.

A.1.8 The `LC_COLLATE` category

Collation controls the relative order of characters and of strings of characters. In general, the ordering of strings and individual characters is independent. However, they are typically closely related.

A.1.8.1 Collation sequence

The relative order of characters is referred to as the *collation sequence*. It defines the characters referred to by a range in regular expressions, such a A-Z or 0-9. The collation sequence is defined by a simple listing of the characters in order, one per line.

Additionally, it is possible to define a multicharacter sequence as having a unique position in the collation sequence. Such a sequence is called a *multicharacter collating element*, whereas the simpler term *collating element* refers to either a character or a multicharacter collating element. For example, the two-character sequence `ch` could be treated as a single character for the purposes of the collation sequence (and for string sorting).

The following is a simple example of a collation sequence:

```
LC_COLLATE
collating-element <ch> from "<c><h>"
collating-element <CH> from "<C><H>"
order_start
<a>
<b>
<c>
<ch>
<d>
<z>
<A>
<B>
<C>
<CH>
<D>
<Z>
<one>
<nine>
order_end
END LC_COLLATE
```

This collation sequence reverses the convention of lowercase preceding uppercase characters. Additionally, it defines uppercase and lowercase forms of the multicharacter collating element `ch`. Also, all digits will succeed alphabetic characters in the collation sequence.

The use of ellipses to indicate ranges of characters is allowed syntax in the locale definition file and is not just a convention for simplifying this example. An ellipsis can also be used before and after the other characters in the collation

sequence to indicate, respectively, all characters earlier or later in the order of the current character encoding, not including the smallest (typically 0) or the largest value.

The keyword `UNDEFINED` can be inserted into the collation sequence. This results in all characters which are not explicitly in the collation sequence being put into the sequence at the point of the `UNDEFINED` statement in the order of their encoded values.

A.1.8.2 String ordering

Character string ordering can also be specified by extending the syntax described in the preceding example. In general, the locale definition file can describe a multipass ordering of strings with pass-specific ordering rules. Passes can scan strings in forward or reverse order.

Multipass sorting works in the following manner. Two strings are compared on the first pass. If they are not equal, the ordering for the first pass defines the ordering of the strings. If, however, they are equal on the first pass, a second comparison pass will be done. This continues until a pass compares the strings as unequal or the maximum number of passes have been executed.

String sorting is defined by the weights of the collating elements being compared. These are specified by putting the weights to the right of the specification of an element in the collation sequence. There may be up to `COLL_WEIGHTS_MAX` (currently 8) weights specified, each separated by a semicolon. A weight can be any of the following:

1. A character. In this case, the order is indicated by the position of that character in the collation sequence.
2. A multicharacter collating element. The order is indicated by the collating elements' position in the collation sequence.
3. A collating symbol. A collation symbol is a symbol that marks a position in the collation sequence. Once defined, the only purpose for a collation symbol is to define weights for collating elements.
4. An ellipsis. In this case, it refers to the collation value of the character or collation element. It is only valid to use this on a line that begins with an ellipsis or in an `UNDEFINED` statement.
5. The keyword `IGNORE`. In this case the collating element is ignored for the purposes of sorting. One exception to this is if the `position` parameter is specified for the associated collation pass.

The following is an example of a specification of a collation sequence with explicit string ordering information:

```
LC_COLLATE
collating-symbol <LOW>
order_start     forward;backward
UNDEFINED
<LOW>
<a>       <a>;<a>
<b>       <b>;<b>
<c>       <c>;<c>
<d>       <d>;<d>
<z>       <z>;<z>
<A>       <a>;<A>
<B>       <b>;<B>
<C>       <c>;<C>
<D>       <d>;<D>
<E>       <e>;<E>
<F>       <f>;<F>
<G>       <g>;<G>
<H>       <h>;<H>
<I>       <i>;<I>
<J>       <j>;<J>
<K>       <k>;<K>
<L>       <l>;<L>
<M>       <m>;<M>
<N>       <n>;<N>
<O>       <o>;<O>
<P>       <p>;<P>
<Q>       <q>;<Q>
<R>       <r>;<R>
<S>       <s>;<S>
<T>       <t>;<T>
<U>       <u>;<U>
<V>       <v>;<V>
<W>       <w>;<W>
<X>       <x>;<X>
<Y>       <y>;<Y>
<Z>       <z>;<Z>
<one>     <one>;<LOW>
...       ...;<LOW>
<nine>    <nine>;<LOW>
END LC_COLLATE
```

The preceding example is case-insensitive on the first pass but considers case on the second pass. For digits they are considered to be higher than alphabetic characters in the first pass and are sorted according to their numeric value. However, in the second pass they will sort after all the alphabetic characters and will be considered equivalent to each other.

The specification of the weights of the lowercase letters is unnecessary since the default for unspecified weights is to use the location of the collating element in the collation sequence.

The previous example is not very useful for any real-world collation. A more typical use of multipass and multidirection sorting would be in the processing of accents or other diacriticals. The first pass would compare two strings in a forward direction without considering the diacriticals. If the strings were equal, the second pass would compare the strings backward considering the diacriticals significant, as in the following example:

```
LC_COLLATE
order_start     forward;backward,position
<a>               <a>;<a>
<a-acute> <a>;<a-acute>
<a-grave> <a>;<a-grave>
<a-circumflex>    <a>;<a-circumflex>
<a-diaeresis>     <a>;<a-diaeresis>
order_end
LC_COLLATE
```

The use of the keyword `position` in describing the second pass is not significant in this example and is added to give an example of the general format of an `order_start` directive. That format is of a semicolon-separated list of pass-specific parameters. When multiple parameters refer to the same pass, they are separated by commas. For example:

```
order_start     forward;backward;forward,position;backward,position
```

The only valid parameters for a pass are the following:

`forward`	The pass shall scan the string from beginning to end.
`backward`	The pass shall scan the string from the end to the beginning.
`position`	The position of ignored weights will be considered significant. The string with the first mismatched ignored element shall succeed the other string.