# Solutions of common problems

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.8 $

# Table of Contents

This library contains the following modules:

| | |
|---|---|
| `Message` | Error Reports (module deleted) |
| `Strings` | String Concatenation |
| `Counter` | Counting Symbol Occurrences |
| `MakeName` | Generating Optional Identifiers |
| `Sort` | Sorting Elements of an Array |

# 1 Error Reports

This module is NOT supported anymore.

Uses of the two macros `Message` and `MessageId` that were provided by this module have to be rewritten, since LIGA does NOT allow to hide accesses of `COORDREF`.

A macro call `Message(s, t)` can be rewritten `message(s, t, 0, COORDREF)`.

A macro call `MessageId(s, t, sym)` can be rewritten

```
message(s, CatStrInd(t, sym), 0, COORDREF)
```

where `CatStrInd` is a macro that stores the concatenation of two strings. It is provided by the module (see Chapter 2 [Strings], page 3):

```
$/Tech/Strings.specs
```

# 2 String Concatenation

This module provides functions for concatenation of strings, and the name `CharPtr` for the type `char *`.

The module is instantiated by

```
$/Tech/Strings.specs
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `Strings.h` is imported there.

The module defines the type name `CharPtr` for `char *`, especially to be used in LIDO specifications.

The module exports the functions

`CharPtr CatStrStr (CharPtr s1, CharPtr s2)`
: The strings `s1, s2` are concatenated. The resulting string is stored in the string memory of the `csm` module, and the pointer to it is returned.

`int IndCatStrStr (CharPtr s1, CharPtr s2)`
: The strings `s1, s2` are concatenated. The resulting string is stored in the string memory of the `csm` module, and its `StringTable` index is returned.

A macro `CatStrInd(s,i)` is exported it obtains the second of the concatenated strings by the index `i` into the `StringTable`. This macro is used to simplify composition of `message` texts.

# 3 Counting Symbol Occurrences

This module provides `.lido` computations that count all ocurrences of certain symbols within a certain subtree.

The module is instantiated by

```
$/Tech/Counter.gnrc +instance=NAME :inst
```

The optional generic instance parameter `NAME` identifies the particular instance.

The module provides the following computational roles:

`NAMECount` is associated to the grammar symbols that shall be counted. `NAMECount.NAMECount` yields the occurrence number in the tree.

`NAMERootCount` specifies the subtree containing the `NAMECount` occurrences. The number of occurrences found in the subtree can be obtained by the attribute `NAMERootCount.NAMECountResult`.

The default is that counting starts from 1 and is incremented by The start value can be adjusted by overriding the computation `NAMERootCount.NAMEInitCount = 0;` with a computation of a suitable value. The increment can be adjusted by overriding the computation of `NAMECount.NAMEIncrement`.

|NAME|RootCount is inherited by the grammar root by default.

`NAMERootCount` can be associated to recursive grammar symbols. Any symbol that has `NAMECount` associated must belong to a subtree such that `NAMERootCount` is associated to its root.

# 4 Generating Optional Identifiers

This C module implements functions that turn their arguments into strings which then play the role of identifiers as if they occurred in the input, i.e. they are entered in the identifier table, and their symbol code is returned.

The module is instantiated by

```
$/Tech/MakeName.gnrc +instance=IDENT :inst
```

The generic instance parameter `IDENT` has to be set to the terminal symbol used for identifiers.

The module can also be used to generate and store identifiers in processors that do not have a scanner and a parser. In this case the `+instance` parameter has to be omitted.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `MakeName.h` is imported there.

The module exports the following functions:

`int MakeName (char *c)`
: `c` is a character string that may coincide with an already existing name. The result the is the encoding of the name `c`.

`int GenerateName (char *c)`
: `c` is a character string. The name is generated by appending a number to the prefix `c` such that the name is different from all others encountered so far (on input and generated).

`int IdnNumb (int id, int num)`
: `id` is the encoding of an existing identifier; `num` is a nonnegative number. The new name is formed by appending the number to the identifier string. It may coincide with an already existing name. This function may be used to derive arbitrary names from existing ones.

`int PreIdnPost (char *pre, int id, char *post)`
: `id` is the encoding of an existing identifier, `pre` and `post` are arbitrary character strings. The generated name is formed by catenation of `pre`, the identifier string and `post`. It

may coincide with an already existing name. This function may be used to derive different names (e. g. for different target objects) from an existing name.

# 5 Sorting Elements of an Array

This C module implements a function that sorts the elements of an array in place. It uses the O(n log n) Heapsort algorithm.

The module is instantiated by

```
$/Tech/Sort.gnrc +instance=TYPE +referto=HDR :inst
```

where `TYPE` is the name of the array element type and `HDR.h` is a file that defines the array element type, e.g.

```
$/Adt/Sort.gnrc+instance=DefTableKey +referto=deftbl :inst
```

If the element type is predefined in C the `referto` parameter is omitted, e.g.

```
$/Adt/Sort.gnrc+instance=int :inst
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `SortTYPE.h` is imported there.

The module exports the following function:

`void SortTYPE (TYPE *arr, size_t n, TYPECmpFctType cmp)`

: `arr` points to the array to be sorted, which contains `n` elements. `cmp` is the function that defines the collating sequence for the elements. It's signature is `TYPE, TYPE -> int`; it yields -1 if the left argument should precede the right in the sorted array, +1 if the left argument should follow the right in the sorted array, and 0 if the order of the elements in the sorted array is immaterial.

# Index