NAME

caesar_cache_1 – the ‘‘cache_1’’ library of OPEN/CAESAR

PURPOSE

The ‘‘cache_1’’ library provides primitives for storing arbitrary data items in caches, which can be organized hierarchically. It can be used to speed up the execution of applications that involve costly search operations (by storing data items in caches for fast retrievals), and also to reduce the memory consumption of state space exploration algorithms (by storing a certain amount of states in caches).

USAGE

The ‘‘cache_1’’ library consists of:

- a predefined header file **`caesar_cache_1.h`**;
- the precompiled library file **`libcaesar.a`**, which implements the features described in **`caesar_cache_1.h`**.

Note: The ‘‘cache_1’’ library is a software layer built above the primitives offered by the ‘‘standard’’, ‘‘area_1’’, and ‘‘hash’’ libraries, and by the *OPEN/CAESAR* graph module. It follows as close as possible the principles and features of the ‘‘table_1’’ library.

DESCRIPTION

A ‘‘cache’’ is basically a set containing a fixed number of items.

Each item in the cache is basically a byte string of fixed size. All items in a given cache have the same size. An item can be considered as a tuple with two fields, whose size and contents are freely determined by the user:

- (1) a ‘‘base’’ field, that is a byte string of fixed size. In a given cache, all base fields have the same size. This size must be greater than zero.

  Sometimes, the base field contains a state (as defined in the graph module). However, this is not mandatory, and base fields can contain other information than states.
- (2) a ‘‘mark’’ field, that is a byte string whose size and contents are freely determined by the user. In a given cache, all mark fields have the same size, which must be greater or equal to zero. Pointers to mark fields will be considered as values of type **`CAESAR_TYPE_POINTER`**; mark fields are always aligned on appropriate boundaries so that the user can put any information in these fields without alignment problem.

The user also determines the nature of the data stored in these fields, which is not meaningful to the ‘‘cache_1’’ library.

A cache is organized as a collection of ‘‘subcaches’’, each one containing a fixed number of items. A cache can contain no more than a maximum of P subcaches, where currently P = 256. In a cache containing N subcaches, each subcache is assigned an unique index in the range 0..N-1. The subcaches are linked hierarchically by a parent relation, which links a subcache to its ‘‘children’’ subcaches. The parent relation is an acyclic relation having as root element the subcache of index 0, denoted ‘‘root subcache’’ in the sequel. All the other subcaches are assumed to be descendants of the root subcache, i.e., they should be reachable from the root subcache via the parent relation. A subcache without children subcaches is called a ‘‘leaf’’ subcache. The parent relation linking the subcaches of a cache is not necessarily static, but can dynamically change during the usage of the cache.

A cache can contain no more than a maximum of M items. Currently, M = 2^{29} = 536,870,912 on 32-bit machines and M = 2^{34} = 17,179,869,184 on 64-bit machines. But, for each cache, the user can also limit the maximal number of items to a lesser bound K < M.

To each cache is associated its current global date, which is the number of modifying operations performed on the cache since it was created. The operations modifying the status of a cache are the following: putting an item into the cache, ‘‘hitting’’ an item (i.e., searching and finding the item) in the cache, deleting an item from the cache, and updating an item present in the cache. In the sequel, whenever this is understood from the context, we will use the term ‘‘date’’ to designate the current global date of a cache. To each subcache of a cache is associated its current local date, i.e., the date when the last modifying operation was performed on an item present in the subcache.

To each item present in a cache are associated the following fields:

- the date when the item was put into the cache;
- the date of the last access to the item by a put, a hit, or an update operation;
- the number of hits at the item since it was put into the cache.

All these fields can be accessed from the address of the item (i.e., a pointer to the memory location where the item is stored in the cache).

Invariant property 1: all items present in a cache have different dates when they were put into the cache. Therefore, the date when an item was put into the cache can serve as unique identification number for the item.

Invariant property 2: all items present in a cache have different last access dates (but several items may have the same number of hits).

Invariant property 3: it is not allowed to modify the base field of any item in the cache. But it is possible to modify the mark field of any item.

Each item of a cache can be accessed by using its address or its base field. The cache data structure establishes a correspondence between these two data. Indeed:

- given an address, one can retrieve the base field and the mark field of the corresponding item;
- given a base field, one can retrieve the address and the mark field of the corresponding item.

Retrieving the address of an item from its base field involves some associative search. To allow fast retrievals, a hash-table is associated to each cache. This is quite transparent from the user’s point of view. Only the base field is taken into account when computing the hash-value and comparing items; the mark field is not meaningful for the search.

To each cache are associated two counters recording the number of search and hit operations performed on the cache, respectively. To each subcache of a cache is associated a counter recording the number of hit operations at (items of) that subcache.

The operation of putting an item E into a cache proceeds as follows. The item is always put into the root subcache (of index 0). When this subcache becomes full, the item E replaces the smallest item E1 contained in this subcache according to the order relation underlying the replacement strategy associated to this subcache. The item E1 is put into one of the child subcaches (of index I1) of the root subcache, determined by the parent relation between subcaches and by the contents of the item E1. If the subcache of index I1 is also full, then the item E1 replaces the smallest item E2 contained in this subcache, and the item E2 will be in turn put into a child subcache (of index I2) of the subcache of index I1, and so on. This process continues along a sequence of subcaches in the hierarchy until either it reaches a subcache of index Ij that is not full (and can therefore accept the current item Ej to be put into it), or it reaches a leaf subcache of index Ik that is full. In the latter case, the item Ek+1 replaced in the leaf subcache is stored temporarily in a field associated to the cache until the next put operation on the cache causes another item El+1 to be replaced in some leaf subcache of index Il that is already full; the item Ek+1 is then deleted and replaced by El+1.

The parent relation between the subcaches of a cache is assumed to be acyclic (in order to ensure the termination of put operations), but this condition cannot be checked at the creation of the cache because the parent relation may change dynamically during execution. Instead, this condition is checked at each put operation, and if a cycle between subcaches is detected, then the operation is stopped and an appropriate error code is set.

The ‘‘cache_1’’ library supports applications involving dynamic creation of caches. When a put operation performed on a cache C1 causes a replacement to take place, another cache C2 can be created and the last item replaced in C1 (which can be inspected using the **CAESAR_LAST_ITEM_REPLACED_CACHE_1()** procedure below) can be put into C2. This enables to build hierarchies of caches similar to the hierarchies of subcaches present in individual caches. It is the user's responsibility to ensure that a put operation performed on a cache of a hierarchy does not cause a cycle of put operations on the other caches of the hierarchy.

FEATURES

....

CAESAR_TYPE_CACHE_1

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_CACHE_1;
```

This type denotes a pointer to the concrete representation of a cache. This representation is supposed to be ‘‘opaque’’.

....

CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1

```
typedef CAESAR_TYPE_NATURAL
   (*CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1) (CAESAR_TYPE_NATURAL);
```

This type denotes a pointer to a function which takes as parameter a natural number (index of a subcache) and returns a natural number (size or percentage of the subcache).

....

CAESAR_TYPE_ORDER_FUNCTION_CACHE_1

```
typedef CAESAR_TYPE_INTEGER
   (*CAESAR_TYPE_ORDER_FUNCTION_CACHE_1) (CAESAR_TYPE_CACHE_1,
      CAESAR_TYPE_POINTER, CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a function which takes as parameters a cache and two pointers to items (supposed to be contained in the cache), and returns an integer number indicating whether the first item is smaller than, equal to, or greater than the second one modulo an order relation.

....

CAESAR_TYPE_SUBCACHE_FUNCTION_CACHE_1

```
typedef CAESAR_TYPE_NATURAL
```

```
      (*CAESAR_TYPE_SUBCACHE_FUNCTION_CACHE_1) (CAESAR_TYPE_CACHE_1,
         CAESAR_TYPE_NATURAL, CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a function which takes as parameters a cache, a natural number (index of a subcache) and a pointer to an item (the last item replaced in the subcache), and returns a natural number (index of the child subcache in which the replaced item will be put).

....

CAESAR_TYPE_CLEANUP_FUNCTION_CACHE_1

```
typedef void
   (*CAESAR_TYPE_CLEANUP_FUNCTION_CACHE_1) (CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a procedure which takes as parameter a pointer to an item and cleans up the contents of the item (see the procedure **CAESAR_CREATE_CACHE_1()** below).

....

CAESAR_TYPE_ERROR_CACHE_1

```
typedef enum {
      CAESAR_NONE_CACHE_1,
      CAESAR_CYCLIC_CACHE_1
}     CAESAR_TYPE_ERROR_CACHE_1;
```

This enumerated type defines the error codes produced as a side effect by calls to the procedure **CAESAR_PUT_CACHE_1()** or to the function **CAESAR_SEARCH_AND_PUT_CACHE_1()** (see below), which put an item into a cache. The error codes have the following meaning:

- **CAESAR_NONE_CACHE_1** indicates that the put operation was performed successfully.
- **CAESAR_CYCLIC_CACHE_1** indicates the existence of a cycle in the parent relation between subcaches, which would cause the put operation to loop indefinitely when the subcaches present on that cycle are full.

Note: The error code produced by a call to the procedure **CAESAR_PUT_CACHE_1()** or to the function **CAESAR_SEARCH_AND_PUT_CACHE_1()** can be obtained by using the function **CAESAR_STATUS_PUT_CACHE_1()** (see below).

....

CAESAR_LRU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least recently used’’ (LRU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date of the last access to the first item is smaller than, equal to, or greater than the date of the last access to the second item, respectively.

....

CAESAR_MRU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most recently used’’ (MRU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date of the last access to the first item is greater than, equal to, or smaller than the date of the last access to the second item, respectively.

....

CAESAR_LRP_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least recently put’’ (LRP) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date when the first item was put into the cache is smaller than, equal to, or greater than the date when the second item was put into the cache, respectively.

....

CAESAR_MRP_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most recently put’’ (MRP) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date when the first item was put into the cache is greater than, equal to, or smaller than the date when the second item was put into the cache, respectively.

....

CAESAR_LFU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LFU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least frequently used’’ (LFU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the number of hits at the first item is smaller than, equal to, or greater than the number of hits at the second item, respectively.

....

CAESAR_MFU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MFU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most frequently used’’ (MFU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the number of hits at the first item is greater than, equal to, or smaller than the number of hits at the second item, respectively.

....

CAESAR_RND_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_RND_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘random’’ (RND) replacement strategy.

This order relation is implemented by computing for every item a random cost, i.e., a natural number randomly generated by taking into account the seed associated to the cache pointed to by **CAESAR_C** (see the procedure **CAESAR_SEED_RND_CACHE_1()** below), the current global date of the cache, and the base field of the item. According to this relation, an item is smaller than, equal to, or greater than another item if the random cost of the first item is smaller than, equal to, or greater than the random cost of the second item, respectively.

....

CAESAR_LFU_LRU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LFU_LRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least frequently used, then least recently used’’ (LFU_LRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one.

In other words, the LFU_LRU replacement strategy consists in applying first the LFU, then the LRU replacement strategies.

....

CAESAR_LFU_MRU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LFU_MRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least frequently used, then most recently used’’ (LFU_MRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one.

In other words, the LFU_MRU replacement strategy consists in applying first the LFU, then the MRU replacement strategies.

....

CAESAR_LFU_LRP_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LFU_LRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least frequently used, then least recently put’’ (LFU_LRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache.

In other words, the LFU_LRP replacement strategy consists in applying first the LFU, then the LRP replacement strategies.

....

CAESAR_LFU_MRP_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_LFU_MRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘least frequently used, then most recently put’’ (LFU_MRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache.

In other words, the LFU_MRP replacement strategy consists in applying first the LFU, then the MRP replacement strategies.

....

CAESAR_MFU_LRU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MFU_LRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most frequently used, then least recently used’’ (MFU_LRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one.

In other words, the MFU_LRU replacement strategy consists in applying first the MFU, then the LRU replacement strategies.

....

CAESAR_MFU_MRU_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MFU_MRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
```

```
    CAESAR_TYPE_CACHE_1 CAESAR_C;
    CAESAR_TYPE_POINTER CAESAR_B1;
    CAESAR_TYPE_POINTER CAESAR_B2;
    { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most frequently used, then most recently used’’ (MFU_MRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one.

In other words, the MFU_MRU replacement strategy consists in applying first the MFU, then the MRU replacement strategies.

....

CAESAR_MFU_LRP_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MFU_LRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
    CAESAR_TYPE_CACHE_1 CAESAR_C;
    CAESAR_TYPE_POINTER CAESAR_B1;
    CAESAR_TYPE_POINTER CAESAR_B2;
    { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most frequently used, then least recently put’’ (MFU_LRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache.

In other words, the MFU_LRP replacement strategy consists in applying first the MFU, then the LRP replacement strategies.

....

CAESAR_MFU_MRP_ORDER_CACHE_1

```
CAESAR_TYPE_INTEGER CAESAR_MFU_MRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by **CAESAR_B1** is smaller than, equal to, or greater than the item pointed to by **CAESAR_B2** according to the order relation underlying the ‘‘most frequently used, then most recently put’’ (MFU_MRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache.

In other words, the MFU_MRP replacement strategy consists in applying first the MFU, then the MRP replacement strategies.

....

CAESAR_SEED_RND_CACHE_1

```
void CAESAR_SEED_RND_CACHE_1 (CAESAR_C, CAESAR_SEED)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_SEED;
   { ... }
```

This procedure initializes the seed for the random number generator associated to the cache pointed to by **CAESAR_C** with the value of **CAESAR_SEED**. The seed is used for computing the random costs associated to the items contained in the subcaches of the cache that are equipped with the RND replacement strategy.

Note: The value of the seed associated to a cache is set by default to 0 when the cache is created by invoking the procedure **CAESAR_CREATE_CACHE_1()** (see below).

....

CAESAR_CREATE_CACHE_1

```
void CAESAR_CREATE_CACHE_1 (CAESAR_C,
                            CAESAR_NUMBER_OF_SUBCACHES,
                            CAESAR_LIMIT_SIZE,
                            CAESAR_SUBCACHE_SIZE,
                            CAESAR_SUBCACHE_PERCENTAGE,
                            CAESAR_SUBCACHE_ORDER,
                            CAESAR_SUBCACHE_CHILD,
                            CAESAR_BASE_AREA,
                            CAESAR_MARK_AREA,
                            CAESAR_HASH_SIZE,
                            CAESAR_PRIME,
                            CAESAR_COMPARE,
                            CAESAR_HASH,
                            CAESAR_PRINT,
                            CAESAR_CLEANUP,
                            CAESAR_INFO)
   CAESAR_TYPE_CACHE_1 *CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_SUBCACHES;
   CAESAR_TYPE_NATURAL CAESAR_LIMIT_SIZE;
   CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1 CAESAR_SUBCACHE_SIZE;
   CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1 CAESAR_SUBCACHE_PERCENTAGE;
   CAESAR_TYPE_ORDER_FUNCTION_CACHE_1 (*CAESAR_SUBCACHE_ORDER)
      (CAESAR_TYPE_NATURAL);
   CAESAR_TYPE_SUBCACHE_FUNCTION_CACHE_1 (*CAESAR_SUBCACHE_CHILD)
      (CAESAR_TYPE_NATURAL);
   CAESAR_TYPE_AREA_1 CAESAR_BASE_AREA;
   CAESAR_TYPE_AREA_1 CAESAR_MARK_AREA;
   CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE;
   CAESAR_PROMOTE_TO_INT (CAESAR_TYPE_BOOLEAN) CAESAR_PRIME;
   CAESAR_TYPE_COMPARE_FUNCTION CAESAR_COMPARE;
   CAESAR_TYPE_HASH_FUNCTION CAESAR_HASH;
   CAESAR_TYPE_PRINT_FUNCTION CAESAR_PRINT;
   CAESAR_TYPE_CLEANUP_FUNCTION_CACHE_1 CAESAR_CLEANUP;
   CAESAR_TYPE_POINTER CAESAR_INFO;
   { ... }
```

This procedure allocates a cache using **CAESAR_CREATE()** and assigns its address to ***CAESAR_C**. If the allocation fails, the **NULL** value is assigned to ***CAESAR_C**.

Note: Because **CAESAR_TYPE_CACHE_1** is a pointer type, any variable **CAESAR_C** of type **CAESAR_TYPE_CACHE_1** must be allocated before used, for instance using:

```
CAESAR_CREATE_CACHE_1 (&CAESAR_C, ...);
```

The value of **CAESAR_NUMBER_OF_SUBCACHES** determines the number of subcaches contained in the cache. Each subcache will be assigned an unique index in the range 0..**CAESAR_NUMBER_OF_SUBCACHES** - 1. If the value of **CAESAR_NUMBER_OF_SUBCACHES** is zero or greater than P, the effect is undefined.

The value of **CAESAR_LIMIT_SIZE** determines the maximal number of items that can be stored in the cache. It must be less or equal to M. If it is equal to zero, it is replaced by the default value M.

Note: in order to spare memory, the value of **CAESAR_LIMIT_SIZE** (which is an upper bound on the number of items to be inserted in the cache) should be as small as possible. This can only be done if the user has some knowledge about the way the cache will be used.

The actual value of the formal parameter **CAESAR_SUBCACHE_SIZE** will be stored and associated to the cache pointed to by ∗**CAESAR_C**. It will be used to assign to each subcache its corresponding size, i.e., the maximal number of items the subcache can contain.

Precisely, the actual value of **CAESAR_SUBCACHE_SIZE** should be a pointer to a function with a parameter **caesar_index** that returns the size of the subcache of index **caesar_index**, where **caesar_index** is in the range 0..**CAESAR_NUMBER_OF_SUBCACHES** - 1. If the value returned by **CAESAR_SUBCACHE_SIZE** for some index **caesar_index** is greater than zero, then the size of the subcache of index **caesar_index** is set to this value. If the value returned is zero, the size of the subcache of index **caesar_index** is unspecified and will be determined by the percentage returned by the **CAESAR_SUBCACHE_PERCENTAGE** parameter (see below). The sum of the sizes returned by **CAESAR_SUBCACHE_SIZE** for all subcaches, noted L, must be less or equal to **CAESAR_LIMIT_SIZE**.

The actual value of the formal parameter **CAESAR_SUBCACHE_PERCENTAGE** will be stored and associated to the cache pointed to by ∗**CAESAR_C**. It will be used to assign to each subcache its corresponding percentage, i.e., a natural number in the range 0..100 determining the size of the subcache with respect to the size of the other subcaches.

Precisely, the actual value of **CAESAR_SUBCACHE_PERCENTAGE** should be a pointer to a function with a parameter **caesar_index** that returns the percentage of the subcache of index **caesar_index**, where **caesar_index** is in the range 0..**CAESAR_NUMBER_OF_SUBCACHES** - 1. Several configurations are possible, enabling the setup of subcache sizes in a flexible manner:

- If the value returned by **CAESAR_SUBCACHE_SIZE** is greater than zero for all subcaches, then the values returned by **CAESAR_SUBCACHE_PERCENTAGE** are ignored because all subcache sizes are determined by **CAESAR_SUBCACHE_SIZE**.
- If there is some subcache with unspecified size (i.e., for which **CAESAR_SUBCACHE_SIZE** returns zero), then the size of the cache pointed to by ∗**CAESAR_C** is set to **CAESAR_LIMIT_SIZE**. Let R be the number of subcaches with unspecified size. The sizes of these subcaches are determined based on the percentages returned by **CAESAR_SUBCACHE_PERCENTAGE**. Two situations are possible:

  (1) If the value returned by **CAESAR_SUBCACHE_PERCENTAGE** is zero for all subcaches with unspecified size, then the size of each of these subcaches is set to (**CAESAR_LIMIT_SIZE** - L) / R.

  (2) If there is some cache with unspecified size for which the value returned by **CAESAR_SUBCACHE_PERCENTAGE** is greater than zero, then the size of a subcache with percentage F is set to (**CAESAR_LIMIT_SIZE** - L) ∗ F / 100. All the percentages returned by **CAESAR_SUBCACHE_PERCENTAGE** for the subcaches with unspecified size must be greater than zero, and the sum of all these percentages must be equal to 100.

The actual value of the formal parameter **CAESAR_SUBCACHE_ORDER** will be stored and associated to the cache pointed to by ∗**CAESAR_C**. It will be used to assign to each subcache a function implementing the order relation underlying the replacement strategy associated to the subcache.

Precisely, the actual value of **CAESAR_SUBCACHE_ORDER** should be a pointer to a function with a parameter **caesar_index** that returns a function implementing an order relation, where **caesar_index** is in the range 0..**CAESAR_NUMBER_OF_SUBCACHES** - 1. The value returned by **CAESAR_SUBCACHE_ORDER** should be a pointer to a function with three parameters **caesar_cache**, **caesar_base_1**, **caesar_base_2**. This function returns a value smaller than, equal to, or greater than 0 if the base field pointed to by **caesar_base_1** is smaller than, equal to, or greater than the base field pointed to by **caesar_base_2**. The items whose base fields are pointed to by **caesar_base_1** and **caesar_base_2** are supposed to be contained in the cache pointed to by **caesar_cache**, which is always set to the value of ***CAESAR_C**, i.e., a pointer to the cache currently created. Examples of functions that can be returned by **CAESAR_SUBCACHE_ORDER** are those implementing the order relations underlying the predefined replacement strategies, namely **CAESAR_LRU_ORDER_CACHE_1()**, **CAESAR_MRU_ORDER_CACHE_1()**, etc.

The actual value of the formal parameter **CAESAR_SUBCACHE_CHILD** will be stored and associated to the cache pointed to by ***CAESAR_C**. It will be used to assign to each subcache a function implementing the parent relation that defines the child subcaches of that subcache.

Precisely, the actual value of **CAESAR_SUBCACHE_CHILD** should be a pointer to a function with a parameter **caesar_index** that returns a function implementing a parent relation, where **caesar_index** is in the range 0..**CAESAR_NUMBER_OF_SUBCACHES** - 1. The value returned by **CAESAR_SUBCACHE_CHILD** should be a pointer to a function with three parameters **caesar_cache**, **caesar_index**, **caesar_base**. The parameter **caesar_cache** is always set to the value of ***CAESAR_C**, i.e., a pointer to the cache currently created. The parameter **caesar_base** is a pointer to the last item replaced in the subcache of index **caesar_index** when the current put operation performed on the cache pointed to by **caesar_cache** entailed a put operation on the subcache of index **caesar_index**, which was already full. This function returns, for the subcache of index **caesar_index**, the index of its child subcache in which the item pointed to by **caesar_base** will be put. If the index returned by this function is greater or equal to **CAESAR_NUMBER_OF_SUBCACHES**, then the subcache of index **caesar_index** is considered to be a leaf subcache, and the last item replaced pointed to by **caesar_base** will be temporarily stored in a field of the cache until the next replacement takes place in the cache (see the **CAESAR_LAST_ITEM_REPLACED_CACHE_1()** procedure below).

For example, the following function implements a parent relation corresponding to a stream of subcaches, i.e., a hierarchy in which each subcache of index I has a single child subcache of index I+1:

```
CAESAR_TYPE_NATURAL caesar_child (caesar_cache, caesar_index, caesar_base)
CAESAR_TYPE_CACHE_1 caesar_cache;
CAESAR_TYPE_NATURAL caesar_index;
CAESAR_TYPE_POINTER caesar_base;
{
   return (caesar_index + 1);
}
```

This function does not use the **caesar_cache** and the **caesar_base** parameters. However, general user-defined functions can implement parent relations that may change dynamically depending on the current status of the cache pointed to by **caesar_cache**, the current status of its subcache of index **caesar_index** and/or the contents of the item pointed to by **caesar_base**.

The value of **CAESAR_BASE_AREA** determines the (constant) size and (constant) alignment factor of the base field in the cache. In the particular case where base fields are used to store states (resp. labels, strings), one must give the value **CAESAR_STATE_AREA_1()** (resp. **CAESAR_LABEL_AREA_1()**, **CAESAR_STRING_AREA_1()**) to the formal parameter **CAESAR_BASE_AREA**.

The value of **CAESAR_MARK_AREA** determines the (constant) size and (constant) alignment factor of the mark field according to the specifications of the ‘‘area_1’’ library. In particular, if **CAESAR_MARK_AREA** is equal to **CAESAR_EMPTY_AREA_1()**, there will be no mark field in the cache.

Each item in the cache will be represented as a byte string of fixed size **caesar_item_size**, such that **caesar_item_size** is greater or equal to **caesar_base_size** + **caesar_mark_size**, where **caesar_base_size** denotes the size (in bytes) of the base field (i.e., **CAESAR_SIZE_AREA_1 (CAESAR_BASE_AREA)**) and where **caesar_mark_size** denotes the size (in bytes) of the mark field, if any (i.e., **CAESAR_SIZE_AREA_1 (CAESAR_MARK_AREA)**).

An item in the cache contains not only the base field and the mark field, but also "padding" bytes that may be inserted between the base and mark fields to ensure that these fields are correctly aligned according to **CAESAR_ALIGNMENT_AREA_1 (CAESAR_BASE_AREA)** and **CAESAR_ALIGNMENT_AREA_1 (CAESAR_MARK_AREA)**.

The value of **CAESAR_HASH_SIZE** determines the number of entries in the hash-table associated to the cache. If it is equal to zero, it is replaced with a default value greater than zero.

If the value of **CAESAR_PRIME** is equal to **CAESAR_TRUE** and if the value of **CAESAR_HASH_SIZE** is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value of **CAESAR_HASH_SIZE** will be kept unchanged.

The actual value of the formal parameter **CAESAR_COMPARE** will be stored and associated to the cache pointed to by ∗**CAESAR_C**. It will be used as a comparison function when it is necessary to decide whether two base fields are equal or not.

Precisely, the actual value of **CAESAR_COMPARE** should be a pointer to a comparison function with two parameters **caesar_base_1** and **caesar_base_2** that returns **CAESAR_TRUE** if the two base fields pointed to by **caesar_base_1** and **caesar_base_2** are equal.

If the actual value of the formal parameter **CAESAR_COMPARE** is **NULL**, it is replaced by a pointer to a default comparison function that depends on the value of **CAESAR_BASE_AREA** and is determined according to the rules specified for function **CAESAR_USE_COMPARE_FUNCTION_AREA_1()** of the "area_1" library.

The actual value of the formal parameter **CAESAR_HASH** will be stored and associated to the cache pointed to by ∗**CAESAR_C**. It will be used as a hash-function when it is necessary to compute a hash-value for searching or inserting an item in the cache.

Precisely, the actual value of **CAESAR_HASH** should be a pointer to a hash function with two parameters **caesar_pointer** and **caesar_modulus** that returns a natural number in the range 0..**caesar_modulus** - 1.

If the actual value of the formal parameter **CAESAR_HASH** is **NULL**, it is replaced by a pointer to a default hash function that depends on the value of **CAESAR_BASE_AREA** and is determined according to the rules specified for function **CAESAR_USE_HASH_FUNCTION_AREA_1()** of the "area_1" library.

The actual value of the formal parameter **CAESAR_PRINT** will be stored and associated to the cache pointed to by ∗**CAESAR_C**. It will be used subsequently to print the items of this cache.

Precisely, the actual value of **CAESAR_PRINT** should be a pointer to a printing procedure with two parameters **caesar_file** and **caesar_item** that prints to file **caesar_file** information about the contents (base field and/or mark field, if any) of the item pointed to by **caesar_item**.

If the actual value of the formal parameter **CAESAR_PRINT** is **NULL**, it is replaced by a pointer to a default procedure that prints the base field and the mark field, if any. The printing procedure used for the base field (respectively, the mark field) depends on the value of **CAESAR_BASE_AREA** (resp. **CAESAR_MARK_AREA**) and is determined according to the rules specified for function **CAESAR_USE_PRINT_FUNCTION_AREA_1()** of the "area_1" library.

The actual value of the formal parameter **CAESAR_CLEANUP** will be stored and associated to the cache pointed to by *`CAESAR_C`. It will be used subsequently to clean up the contents of the items deleted from this cache.

Precisely, the actual value of **CAESAR_CLEANUP** should be a pointer to a procedure with one parameter `caesar_item` that cleans up the contents (base field and mark field, if any) of the item pointed to by `caesar_item`. This cleanup operation is useful if the base and/or the mark fields (if any) of items contain pointers to dynamic data structures (e.g., lists, sets, etc.) that must be freed when those items are deleted from the cache. For example, if the base field is a list of edges of type **CAESAR_TYPE_EDGE** that must not be kept in memory when items are deleted from the cache, a good candidate for the **CAESAR_CLEANUP** parameter is the `CAESAR_DELETE_EDGE_LIST()` procedure of the ‘‘edge’’ library, which deletes the list pointed to by the base field of the item pointed to by `caesar_item`, and then sets this base field to **NULL**.

If the actual value of the formal parameter **CAESAR_CLEANUP** is **NULL**, there will be no cleanup operation performed on the contents of the items when these items are deleted from the cache.

The actual value of the formal parameter **CAESAR_INFO** will be stored and associated to the cache pointed to by *`CAESAR_C`. This value should be a pointer to some data structure containing user-defined information that will be associated to this cache. The value of this pointer remains unchanged during the lifetime of the cache and can be inspected using the `CAESAR_INFO_CACHE_1()` function (see below).

....

CAESAR_CURRENT_CACHE_1

```
CAESAR_TYPE_CACHE_1 CAESAR_CURRENT_CACHE_1 ()
    { ... }
```

This function returns a pointer to the cache which is currently in use. It should be called only within the functions and procedures given as actual values for the formal parameters **CAESAR_SUBCACHE_SIZE**, **CAESAR_SUBCACHE_PERCENTAGE**, **CAESAR_COMPARE**, **CAESAR_HASH**, **CAESAR_PRINT**, and **CAESAR_CLEANUP** of procedure `CAESAR_CREATE_CACHE_1()` (see above); in this case, the result is a pointer to the cache created by the call to `CAESAR_CREATE_CACHE_1()`. If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to identify the cache to which the items passed as arguments to the six aforementioned functions and procedures belong, and thus to handle these items accordingly (the size and the contents of items belonging to different caches may differ). It is especially useful when the number of caches is unknown statically (e.g., when new caches are created dynamically during the execution of the application program).

....

CAESAR_CURRENT_SUBCACHE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_SUBCACHE_CACHE_1 ()
    { ... }
```

This function returns the index of the subcache which is currently in use; this subcache is in turn contained in the cache which is currently in use, pointed to by the result of function `CAESAR_CURRENT_CACHE_1()` (see above). It should be called only within the functions and procedures given as actual values for the formal parameters **CAESAR_SUBCACHE_ORDER**, **CAESAR_COMPARE**, **CAESAR_HASH**, **CAESAR_PRINT**, and **CAESAR_CLEANUP** of procedure `CAESAR_CREATE_CACHE_1()` (see above); in this case, the result is the index of the subcache, i.e., a natural number in the range 0..N-1, where N is the number of subcaches in the cache created by the call to **CAESAR_CREATE_CACHE_1()**. If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to identify the subcache to which the items passed as arguments to the five aforementioned functions and procedures belong, and thus to handle these items accordingly.

....

CAESAR_DELETE_CACHE_1

```
void CAESAR_DELETE_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 *CAESAR_C;
   { ... }
```

This procedure frees the memory space corresponding to the cache pointed to by ***CAESAR_C** using **CAESAR_DELETE()**. All the items currently present in the cache are freed by invoking first the cleanup function (if any) associated to the cache, and then **CAESAR_DELETE()**. Afterwards, the **NULL** value is assigned to ***CAESAR_C**.

....

CAESAR_PURGE_SUBCACHE_CACHE_1

```
void CAESAR_PURGE_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This procedure reinitializes the information associated to the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**. All the items currently present in the subcache are freed by invoking first the cleanup function (if any) associated to the cache, and then **CAESAR_DELETE()**. Afterwards, the subcache is exactly in the same state as after the creation of the cache using **CAESAR_CREATE_CACHE_1()**.

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the effect is undefined.

....

CAESAR_PURGE_CACHE_1

```
void CAESAR_PURGE_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This procedure reinitializes the information associated to the cache pointed to by **CAESAR_C**. All the items currently present in the cache are freed by invoking first the cleanup function (if any) associated to the cache, and then **CAESAR_DELETE()**. Afterwards, the cache is exactly in the same state as after its creation using **CAESAR_CREATE_CACHE_1()**.

....

CAESAR_SEARCH_CACHE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_CACHE_1 (CAESAR_C, CAESAR_B, CAESAR_N, CAESAR_P)
```

```
    CAESAR_TYPE_CACHE_1 CAESAR_C;
    CAESAR_TYPE_POINTER CAESAR_B;
    CAESAR_TYPE_NATURAL *CAESAR_N;
    CAESAR_TYPE_POINTER *CAESAR_P;
    { ... }
```

This function determines if there exists, in the cache pointed to by **CAESAR_C**, an item whose base field is equal to the byte string pointed to by **CAESAR_B**. Byte string comparisons are performed using the comparison function associated to the cache. The search is done using the hash-function and hash-table associated to the cache.

If so, this function returns **CAESAR_TRUE**. In this case, the index of the subcache containing the existing item and the address of the item are respectively assigned to ***CAESAR_N** and ***CAESAR_P**. The number of searches and hits at the cache and the current global date of the cache are incremented. The number of hits at the subcache is incremented and the current local date of the subcache is set to the current global date of the cache. The number of hits at the item is also incremented and the date of the last access to the item is set to the current global date of the cache (the item becomes the most recently accessed item in the cache).

If not, this function returns **CAESAR_FALSE**. In this case, both variables ***CAESAR_N** and ***CAESAR_P** are left unchanged. The number of searches in the cache is incremented. The number of hits at the cache and the current global date of the cache are left unchanged.

....

CAESAR_PUT_BASE_CACHE_1

```
CAESAR_TYPE_POINTER CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)
    CAESAR_TYPE_CACHE_1 CAESAR_C;
    { ... }
```

This function returns a pointer to the base field of the next item to be put into the cache pointed to by **CAESAR_C**.

The base field pointed to by the result of **CAESAR_PUT_BASE_CACHE_1()** is initially undefined and must be assigned before calling some other functions of the “cache_1” library (see below).

....

CAESAR_PUT_MARK_CACHE_1

```
CAESAR_TYPE_POINTER CAESAR_PUT_MARK_CACHE_1 (CAESAR_C)
    CAESAR_TYPE_CACHE_1 CAESAR_C;
    { ... }
```

This function returns a pointer to the mark field of the next item to be put into the cache pointed to by **CAESAR_C**. If there are no mark fields in the cache (due to the initialization parameters supplied to **CAESAR_CREATE_CACHE_1()**) the result is undefined.

The mark field pointed to by the result of **CAESAR_PUT_MARK_CACHE_1()** is always initialized to a bit string of 0’s. It can be either consulted or modified.

....

CAESAR_PUT_CACHE_1

```
void CAESAR_PUT_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This procedure puts into the cache pointed to by **CAESAR_C** the item whose base field is pointed to by **CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)** and whose mark field (if any) is pointed to by **CAESAR_PUT_MARK_CACHE_1 (CAESAR_C)**.

The base field must have been assigned before this procedure is called.

This procedure also sets a field of type **CAESAR_TYPE_ERROR_CACHE_1** associated to the cache, indicating whether the put operation was carried out successfully or not; this field can be inspected using the function **CAESAR_STATUS_PUT_CACHE_1()** (see below).

The hash-table associated to the cache is updated to take into account the new item. To compute the hash-value for the base field, the hash-function associated to the cache is used.

If the put operation causes another item E contained in (some leaf subcache of) the cache to be replaced, this item is stored temporarily in a field associated to the cache until a future call to **CAESAR_PUT_CACHE_1()** or **CAESAR_SEARCH_AND_PUT_CACHE_1()** will cause another replacement to take place. Meanwhile, the item E can be inspected by using the **CAESAR_LAST_ITEM_REPLACED_CACHE_1()** procedure (see below). When the next replacement takes place, if the item E has not been inspected meanwhile by a call to **CAESAR_LAST_ITEM_REPLACED_CACHE_1()**, it will be first cleaned up using the cleanup function (if any) associated to the cache, and then freed using **CAESAR_DELETE()**; otherwise, the item E will not be cleaned up (because it is the user's responsibility to manage the memory possibly referenced in the contents of this item), but freed using **CAESAR_DELETE()** only.

Note: the cache is implemented in such a way that if a memory shortage occurs during a put operation when the cache is not already full, all the subcaches are considered to become full and their associated replacement strategies start to be used.

....

CAESAR_SEARCH_AND_PUT_CACHE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_AND_PUT_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_P)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL *CAESAR_N;
   CAESAR_TYPE_POINTER *CAESAR_P;
   { ... }
```

This function is a combination of the function **CAESAR_SEARCH_CACHE_1()** and the procedure **CAESAR_PUT_CACHE_1()** defined above.

The base field pointed to by **CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)** must have been assigned before this function is called.

It first determines if there exists, in the cache pointed to by **CAESAR_C**, an item whose base field is equal to the base field of the item pointed to by **CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)**. Byte string comparisons are performed using the comparison function associated to the cache. The search is done using the hash-function and hash-table associated to the cache.

If so, this function returns **CAESAR_TRUE**. In this case, the index of the subcache containing the existing item and the address of the item are respectively assigned to ∗**CAESAR_N** and ∗**CAESAR_P**. The number of searches and hits at the cache and the current global date of the cache are incremented. The number of hits at the subcache is incremented and the current local date of the subcache is set to the current global date of the cache. The number of hits at the item is also incremented and the date of the last access to the item is set to the current global date of the cache (the item becomes the most recently accessed item in the cache). The field of type **CAESAR_TYPE_ERROR_CACHE_1** associated to the cache is set to **CAESAR_NONE_CACHE_1**.

If not, this function returns **CAESAR_FALSE**. In this case, it puts into the cache pointed to by **CAESAR_C** the item whose base field is pointed to by **CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)** and whose mark field (if any) is pointed to by **CAESAR_PUT_MARK_CACHE_1 (CAESAR_C)**. The hash-table associated to the cache is updated to take into account the new item. The number of searches in the cache and the current global date of the cache are incremented.

The field of type **CAESAR_TYPE_ERROR_CACHE_1** associated to the cache is set to indicate whether the put operation was carried out successfully or not. If the put operation succeeded, variable ∗**CAESAR_N** is assigned the value 0 (since the item was put into the root subcache, of index 0) and variable ∗**CAESAR_P** is assigned the address of the item, which is now contained in the cache. If the put operation failed (because a cycle was detected in the parent relation between subcaches), the variables ∗**CAESAR_N** and ∗**CAESAR_P** are left unchanged.

If the put operation causes another item E contained in (some leaf subcache of) the cache to be replaced, this item is stored temporarily in a field associated to the cache until a future call to **CAESAR_PUT_CACHE_1()** or **CAESAR_SEARCH_AND_PUT_CACHE_1()** will cause another replacement to take place. Meanwhile, the item E can be inspected by using the **CAESAR_LAST_ITEM_REPLACED_CACHE_1()** procedure (see below). When the next replacement takes place, if the item E has not been inspected meanwhile by a call to **CAESAR_LAST_ITEM_REPLACED_CACHE_1()**, it will be first cleaned up using the cleanup function (if any) associated to the cache, and then freed using **CAESAR_DELETE()**; otherwise, the item E will not be cleaned up (because it is the user's responsibility to manage the memory possibly referenced in the contents of this item), but freed using **CAESAR_DELETE()** only.

Note: the cache is implemented in such a way that if a memory shortage occurs during a put operation when the cache is not already full, all the subcaches are considered to become full and their associated replacement strategies start to be used.

....

CAESAR_STATUS_PUT_CACHE_1

```
CAESAR_TYPE_ERROR_CACHE_1 CAESAR_STATUS_PUT_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the status of the last put operation performed by a call to the procedure **CAESAR_PUT_CACHE_1()** or to the function **CAESAR_SEARCH_AND_PUT_CACHE_1()** (see above) on the cache pointed to by **CAESAR_C**.

....

CAESAR_MINIMAL_ITEM_CACHE_1

```
CAESAR_TYPE_POINTER CAESAR_MINIMAL_ITEM_CACHE_1 (CAESAR_C, CAESAR_N)
```

```
      CAESAR_TYPE_CACHE_1 CAESAR_C;
      CAESAR_TYPE_NATURAL CAESAR_N;
      { ... }
```

This function returns the address of the smallest item contained in the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**. The smallest item is determined according to the order relation underlying the replacement strategy of the subcache of index **CAESAR_N**.

If the index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache) or the subcache of index **CAESAR_N** is empty, the result is undefined.

....

CAESAR_DELETE_ITEM_CACHE_1

```
   void CAESAR_DELETE_ITEM_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_B)
      CAESAR_TYPE_CACHE_1 CAESAR_C;
      CAESAR_TYPE_NATURAL CAESAR_N;
      CAESAR_TYPE_POINTER *CAESAR_B;
      { ... }
```

This procedure deletes, for the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**, the item whose base field is pointed to by ∗**CAESAR_B**. The item is freed by invoking first the cleanup function (if any) associated to the cache, and then **CAESAR_DELETE()**. Afterwards, the **NULL** value is assigned to ∗**CAESAR_B**.

The number of items in the subcache of index **CAESAR_N** and in the cache is decremented. The current global date of the cache is incremented and the current local date of the subcache of index **CAESAR_N** is set to the current global date of the cache.

If no item stored in the subcache of index **CAESAR_N** of the cache has a base field at address **CAESAR_B**, the effect is undefined.

If the index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache) or the subcache of index **CAESAR_N** is empty, the effect is undefined.

....

CAESAR_LAST_ITEM_REPLACED_CACHE_1

```
   void CAESAR_LAST_ITEM_REPLACED_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_B)
      CAESAR_TYPE_CACHE_1 CAESAR_C;
      CAESAR_TYPE_NATURAL *CAESAR_N;
      CAESAR_TYPE_POINTER *CAESAR_B;
      { ... }
```

This procedure respectively assigns to the variables ∗**CAESAR_N** and ∗**CAESAR_B** the index of the subcache and the address of the item that was replaced in that subcache when the last call to **CAESAR_PUT_CACHE_1()** or to **CAESAR_SEARCH_AND_PUT_CACHE_1()** was performed on the cache pointed to by **CAESAR_C** and caused a replacement to take place.

This procedure also sets an internal field of the cache indicating that the last item replaced was inspected, and from now on it is the user's responsibility to manage the memory possibly referenced in the contents of the item pointed to by ∗**CAESAR_B**. Thus, when some future call to **CAESAR_PUT_CACHE_1()** or to

CAESAR_SEARCH_AND_PUT_CACHE_1() on the cache will cause another item to be replaced, the item pointed to by ***CAESAR_B** will not be cleaned up by invoking the cleanup function (if any) associated to the cache, but its contents will be freed by invoking **CAESAR_DELETE()** only.

If none of the previous calls to **CAESAR_PUT_CACHE_1()** or to **CAESAR_SEARCH_AND_PUT_CACHE_1()** caused a replacement to take place, then the values N (where N is the number of subcaches in the cache) and **NULL** are respectively assigned to the variables ***CAESAR_N** and ***CAESAR_B**.

Note: The item pointed to by ***CAESAR_B** can be handled in the same way as an ordinary item present in the cache, e.g., the address of its mark field can be retrieved using the **CAESAR_RETRIEVE_B_M_CACHE_1()** procedure.

....

CAESAR_UPDATE_CACHE_1

```
void CAESAR_UPDATE_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This procedure simulates, for the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**, a hit at the item whose base field is pointed to by **CAESAR_B**.

The current global date of the cache is incremented and the current local date of the subcache of index **CAESAR_N** is set to the current global date of the cache. The date of the last access to the item pointed to by **CAESAR_B** is set to the current global date of the cache (the item becomes the most recently accessed item in the cache). The number of hits at the item is incremented. Finally, the subcache is updated depending whether the item has become smaller or greater according to the order relation underlying the replacement strategy associated to the subcache.

If no item stored in the subcache of index **CAESAR_N** of the cache has a base field at address **CAESAR_B**, the effect is undefined.

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the effect is undefined.

Note: If the items of the cache contain mark fields, and if the subcache is equipped with a user-defined replacement strategy whose underlying order relation depends on the contents of mark fields, this procedure should be called after any modification of the mark field of an item of the subcache in order to bring the subcache to a consistent state.

....

CAESAR_CURRENT_DATE_SUBCACHE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_DATE_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the date of the last modifying operation performed on an item present in the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**.

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the result is undefined.

....

CAESAR_CURRENT_DATE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_DATE_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the current global date of the cache pointed to by **CAESAR_C**.

....

CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the number of items currently contained in the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**.

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the result is undefined.

....

CAESAR_NUMBER_OF_ITEMS_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_ITEMS_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the number of items currently contained in the cache pointed to by **CAESAR_C**.

....

CAESAR_NUMBER_OF_SEARCHES_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_SEARCHES_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the number of search operations performed on the cache pointed to by **CAESAR_C**.

....

CAESAR_NUMBER_OF_HITS_SUBCACHE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_HITS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the number of hits at the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C**.

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the result is undefined.

....

CAESAR_NUMBER_OF_HITS_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_HITS_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the number of hits at the cache pointed to by **CAESAR_C**.

....

CAESAR_ITEM_PUT_DATE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_ITEM_PUT_DATE_CACHE_1 (CAESAR_C, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This function returns the date when the item whose base field is pointed to by **CAESAR_B** was put into the cache pointed to by **CAESAR_C**.

If no item stored in (some subcache of) the cache has a base field at address **CAESAR_B**, the result is undefined.

....

CAESAR_ITEM_CURRENT_DATE_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_ITEM_CURRENT_DATE_CACHE_1 (CAESAR_C, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This function returns, for the cache pointed to by **CAESAR_C**, the date of the last access to the item whose base field is pointed to by **CAESAR_B**.

If no item stored in (some subcache of) the cache has a base field at address **CAESAR_B**, the result is undefined.

....

CAESAR_ITEM_NUMBER_OF_HITS_CACHE_1

```
CAESAR_TYPE_NATURAL CAESAR_ITEM_NUMBER_OF_HITS_CACHE_1 (CAESAR_C, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This function returns, for the cache pointed to by **CAESAR_C**, the number of hits at the item whose base field is pointed to by **CAESAR_B**.

If no item stored in (some subcache of) the cache has a base field at address **CAESAR_B**, the result is undefined.

....

CAESAR_EMPTY_SUBCACHE_CACHE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns a value different from 0 if the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C** is empty, and 0 otherwise. **CAESAR_EMPTY_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)** is always equivalent to:

```
CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N) == 0
```

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the result is undefined.

....

CAESAR_EMPTY_CACHE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns a value different from 0 if the cache pointed to by **CAESAR_C** is empty, and 0 otherwise. **CAESAR_EMPTY_CACHE_1 (CAESAR_C)** is always equivalent to:

```
CAESAR_NUMBER_OF_ITEMS_CACHE_1 (CAESAR_C) == 0
```

....

CAESAR_FULL_SUBCACHE_CACHE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_FULL_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns a value different from 0 if the subcache of index **CAESAR_N** of the cache pointed to by **CAESAR_C** is full, and 0 otherwise. **CAESAR_FULL_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)** is always equivalent to:

```
CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N) == $K$
```

where K denotes the maximum number of items that the subcache can contain.

If the subcache index **CAESAR_N** is outside the range 0..N-1 (where N is the number of subcaches in the cache), the result is undefined.

....

CAESAR_FULL_CACHE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_FULL_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns a value different from 0 if the cache pointed to by **CAESAR_C** is full, and 0 otherwise. **CAESAR_FULL_CACHE_1 (CAESAR_C)** is always equivalent to:

```
CAESAR_NUMBER_OF_ITEMS_CACHE_1 (CAESAR_C) == $K$
```

where K denotes the maximum number of items that the cache can contain.

....

CAESAR_INFO_CACHE_1

```
CAESAR_TYPE_POINTER CAESAR_INFO_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the pointer to the user information associated to the cache pointed to by **CAESAR_C** when this cache was created using **CAESAR_CREATE_CACHE_1()**. Precisely, the result returned by this function is the value of the formal parameter **CAESAR_INFO** supplied at the call to **CAESAR_CREATE_CACHE_1()**.

....

CAESAR_RETRIEVE_B_M_CACHE_1

```
void CAESAR_RETRIEVE_B_M_CACHE_1 (CAESAR_C, CAESAR_B, CAESAR_M)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes, for the cache pointed to by **CAESAR_C**, the address of the mark field of the item whose base field is pointed to by **CAESAR_B**. This address is assigned to ***CAESAR_M**.

If no item stored in (some subcache of) the cache has a base field at address **CAESAR_B**, the effect is undefined.

If there are no mark fields in the cache (due to the initialization parameters supplied to **CAESAR_CREATE_CACHE_1()**), the effect is undefined.

....

CAESAR_RETRIEVE_M_B_CACHE_1

```
void CAESAR_RETRIEVE_M_B_CACHE_1 (CAESAR_C, CAESAR_M, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_M;
   CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure computes, for the cache pointed to by **CAESAR_C**, the address of the base field of the item whose mark field is pointed to by **CAESAR_M**. This address is assigned to ***CAESAR_B**.

If no item stored in (some subcache of) the cache has a mark field at address **CAESAR_M**, the effect is undefined.

If there are no mark fields in the cache (due to the initialization parameters supplied to **CAESAR_CREATE_CACHE_1()**), the effect is undefined.

....

CAESAR_FORMAT_CACHE_1

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_CACHE_1 (CAESAR_C, CAESAR_FORMAT)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the cache pointed to by **CAESAR_C** will be printed by the procedure **CAESAR_PRINT_CACHE_1()** (see below). Currently, the following formats are available:

- With format 0, statistical information concerning the cache is displayed such as: the number of items, the replacement strategy and the number of hits for each subcache of the cache, the total number of searches and hits for the whole cache, etc.
- (no other format available yet)

By default, the current format of each cache is initialized to 0.

When called with **CAESAR_FORMAT** between 0 and 0, this fonction sets the current format of **CAESAR_C** to **CAESAR_FORMAT** and returns an undefined result.

When called with another value of **CAESAR_FORMAT**, this function does not modify the current format of **CAESAR_C** but returns a result defined as follows. If **CAESAR_FORMAT** is equal to the constant **CAESAR_CURRENT_FORMAT**, the result is the value of the current format of **CAESAR_C**. If **CAESAR_FORMAT** is equal to the constant **CAESAR_MAXIMAL_FORMAT**, the result is the maximal format value (i.e., 0). In all other cases, the effect of this function is undefined.

....

CAESAR_MAX_FORMAT_CACHE_1

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_CACHE_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the *OPEN/CAESAR*. Use function **CAESAR_FORMAT_CACHE_1()** instead, called with argument **CAESAR_MAXIMAL_FORMAT**.

This function returns the maximal format value available for printing caches.

....

CAESAR_PRINT_CACHE_1

```
void CAESAR_PRINT_CACHE_1 (CAESAR_FILE, CAESAR_C)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This procedure prints on file **CAESAR_FILE** an ASCII text containing various informations about the cache pointed to by **CAESAR_C**. The nature of these informations is determined by the current format of the cache pointed to by **CAESAR_C**.

Before this procedure is called, **CAESAR_FILE** must have been properly opened, for instance using **fopen(3)**.

....

AUTHOR(S)

Radu Mateescu

FILES

$CADP/incl/caesar_graph.h	interface of the graph module
$CADP/incl/caesar_*.h	interfaces of the storage module
$CADP/bin.‘arch‘/libcaesar.a	object code of the storage module
$CADP/src/open_caesar/*.c	source code of various exploration modules
$CADP/com/lotos.open	shell script to run OPEN/CAESAR

SEE ALSO

Reference Manuals of OPEN/CAESAR, CAESAR, and CAESAR.ADT, **lotos.open**(LOCAL), **caesar**(LOCAL), **caesar.adt**(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_*.**

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY.**

BUGS

Known bugs are described in the Reference Manual of OPEN/CAESAR. Please report new bugs to cadp@inria.fr