The buffer command creates a type and associated functions for a buffer object.

```
 type: buffer(elt_type,block_size);
```

Here, elt_type is the type of the objects to be stored in the buffer. block_size is the size of each block for the buffer. The buffers are allocated in block_size increments.

The generated functions are:

```
  type*    pfx_new();
  void     pfx_append(type* buffer, elt_type data);
  elt_type* pfx_collect(type *buffer, int *len);
  elt_type* pfx_collectd(type *buffer, int *len);
  void     pfx_free(type *buffer);
```

Here, pfx is equal to type, but the trailing _t is cut off if there is one. So, if you have:

```
  intbuffer_t : buffer(int,50);
```

You would get functions like: *intbuffer_new(), intbuffer_collect()* etc. The *collect* function returns an array of elements whose size is stored in **len* if *len* is not null. *pfx_collectd* is the same as *pfx_collect*, but calls *pfx_free* on the buffer object after the collection.

The collect function returns null if the buffer was empty. You can test for either len==0 or collect()==null, they are both true in this case.

The quick_sort command generates some code to sort the given elements. It's not generated as a separate function. It simply generates some iterative stuff with a limited stack size. The stack is big enough to sort 2^32 elements, i.e. maximum number of elements you can have on a 32 bit machine. It does this by always pushing the larger segment on the stack after partitioning. The syntax is:

```
quick_sort(type, array, length)
 { QSRESULT= compare(array[QSI], array[QSP]); }
```

Here, type is the type of the elements to be sorted. array is a pointer or array value to be sorted and length should be an integer with no side effects. In the comparison code, the variables *QSRESULT*, *QSI* and *QSP* should be used as shown, they are not place holders. The compare function shown above can in fact be any expression comparing element QSI to element QSP. QSRESULT shall be set to a negative value if element QSI is less than element QSP, zero if they are equal and a positive value otherwise. The sort normally sorts in ascending order. You can change the sense of comparison if you want descending order.

In the generated code, there are some generated variables starting with the prefix *yyqr*. Make sure you don't use this weird prefix for your own things visible in the quicksort code.

Unify

unify(array, length, var) { UAR = compare(array[UAA], array[UAB]); }

stores the final length of the array in "var", which must be declared by the user.

The code inside the curly braces should compare element UAA to element UAB and set UAR to non-zero iff the elements are different. UAA, UAB and UAR must not be used in another context.

The stack code generator takes a type as an argument:

```
name: stack(base_type);
```

Here, name is used for the generated type and the function prefixes. base_type indicates the type of the elements. The following struct and functions are created:

```
typedef struct {
        int selts, tos;
        base_type *elts;
} name_t;

name_t *name_new(int initial);
int name_empty(name_t *S);
void name_push(name_t *S, base_type V);
int name_pop(name_t *S, base_type* R);
```

The argument *initial* indicates how many elements shall be allocated initially. Element *tos* of the stack is -1 when the stack is empty. Otherwise, *elts[tos]* gives the top element on the stack. pop returns 0 if the stack is empty. Otherwise, it returns 1 and stores the top element in **R* if *R* is not null. The *elts* array can be freed with *free()*.