```
typedef struct dncomp {
    RDN                 dn_rdn;
    struct dncomp       *dn_parent;
} dncomp, *DN;

#define NULLDN ((DN) 0)
```

The following routines are provided in addition to the routines defined in Section 16.1 to allow the user to create and manipulate names;

- Create a DN given an RDN:

```
dn_comp_new (rdn)
RDN     rdn;
```

- Append a DN to the end of another DN:

```
dn_append (dn1, dn2)
DN      dn1,
        dn2;
```

`dn2` is added to `dn1`.

- Create an RDN given an AttributeType and AttributeValue:

```
RDN rdn_comp_new (at, av)
AttributeType  at;
AttributeValue av;
```

- Merge two RDNs:

```
RDN rdn_merge (rdn1, rdn2)
RDN     rdn1,
        rdn2;
```

RDNs form a set. To make comparisons easier, this implementation assumes them to be ordered sets. This routine takes two ordered RDN sets, and merges them. If RDNs received across the network are not ordered, the decoding process will order them for you.

16.3.1 User-Friendly Naming

There are various routines to implement user friendly naming (UFN)[Kil90]. These are briefly described in this section.

To print a DN to a PStream as a UFN you can use

```
ufn_dn_print (ps,dn,multiline)
PS  ps;
DN  dn;
int multiline;
```

If the `multiline` option is `TRUE` the UFN will be printed on multiple lines, otherwise, a single line of output will be used. The `ufn_dn_print` routine prints each RDN using

```
ufn_rdn_print (ps,rdn)
RDN  rdn;
PS   ps;
```

Parsing a UFN into a DN is a little more complex:

```
ufn_match (argc,argv,interact,result,el)
int argc;
char ** argv;
DNS (* interact) ();
DNS * result;
envlist el;
```

The `argc` and `argv` parameters provide the components of the UFN, which will typically have been generated from a comma separated UFN string. The `el` component is used to pass a UFN environment list to the lookup routine. If this is left as NULL, this is generated from the system wide *ufnrc* file.

A sequence of DNs is returned in the `result` paramenter. `DNS` has the following structure:

```
typedef struct dn_seq {
    DN  dns_dn;
    struct dn_seq *dns_next;
} *DNS;
```

The `interact` parameter if the name of a routine to interact with the user if the UFN process is unclear as to which name to choose.

```
DNS interact (dns,dn,s)
DNS dns;
DN dn;
char * s;
```

The `dns` parameter is the sequence of potential names, `dn` is the DN of the environment below which the names have been, `s` is UFN element being matched. The user should be asked to select which DNs should be used, and the DNs returned. DNs not required in the `dns` passed in should be freed. If user interaction is not required, a simple interact routine to return a NULL DNS should be provided.

16.4 Adding New Syntaxes to QUIPU

It will probably be necessary to add knowledge of extra syntaxes to directory user interfaces as the range of applications grow. QUIPU has been designed to make this as easy as possible.

This section describes how you might do this, by adding knowledge of a `FOOBAR` syntax to *libdsap*(3n) library. `FOOBAR` is represented by the *C* structure "`struct foobar {...};`".

The DSA does not need to know about the new structure, as it will handle unknown syntaxes as blocks of ASN.1.

The procedure call

```
short add_attribute_syntax (sntx,enc,dec,parse,
                        print,cpy,cmp,sfree,
                        print_pe,approx,multiline)
char *  sntx;
IFP     enc,dec,parse,print,cpy,cmp,sfree,approx;
char *  print_pe;
char    multiline;
```

is used to add knowledge of new attribute syntaxes to QUIPU.

The procedure returns a *short* which can be used to reference the syntax handler defined.

The parameters are used as follows:

`sntx`: The string defining the syntax; in this case it should be `Foobar`. Any attributes defined as having the syntax "Foobar" in the oidtables will then be handled using the following routines.

`enc`: This is a function pointer to a routine that will convert a *C* structure representation of foobar into a presentation element that can be sent across the network. The routine is given a single parameter, a pointer to the foobar structure, and is expected to return a PE. You are reminded of the *pepy*(1c), *posy*(1c) and *pepsy*(1c) utilities described in *Volume Four* of this manual, they may be of help in creating this encoder.

`dec`: This performs the inverse of the above, it is passed a PE, and should return a *C* structure representation.

`parse`: Each syntax needs to have a representation that can be read from and written to an EDB file. This defined routine should take a single `char *` argument and return a C structure representation.

`print`: This routine is used to print the syntax to a `PStream`[1]. The arguments are:

```
foobar_print (ps,fb,format)
PS ps;
struct foobar * fb;
int format;
```

If `format` is `READOUT` the structure should be printed to the stream PS in a "human readable" manner; otherwise is should be printed in a form that can be parsed by the "parse" function. *Note*: in many case both formats will be the same.

`cpy`: This function should take a foobar structure as a parameter and returns a copy of it.

`cmp`: A function that takes two foobar structures and compares them; returning 0 if they are the same; 1 if the first is considered "greater" than the second; –1 otherwise. If there is no appropriate ordering

[1] `PStreams` are discussed in *Volume One* of this manual.

for the syntax, returns 2. If an error occurs during the comparison, returns –2.

`sfree:` A routine to free the structure foobar.

`print_pe:` The name of an external process to "exec" when the `READOUT` option is supplied to the print routine. This is generally set to `NULLCP` which means the default "print" routine is used.

`approx:` A routine to perform approximate matching (if required). The routine should expect two parameters as shown:

```
foobar_approx (a,b)
    struct filter_item * a;
    AV_Sequence b;
```

The routine should return `OK` or `NOTOK` depending on whether any attribute value in the `AV_Sequence` approximately matches the `filter_item`.

`multiline:` If `TRUE` the multivalue attributes will be printed on separate lines:

```
foobar = value1
foobar = value2
```

otherwise the attribute will be printed on one line:

```
foobar = value1 & value2
```

16.4.1 Where to Add the Syntax Definition

Where should you add this procedure call to define the new syntaxes?

This depends on which applications you want to know about the syntax. If you want the DSA and all DUAs to know about the syntax, then this call should be added to *libdsap*, in the directory *dsap/common*. The file *dsap/common/quipu_sntx.c* is where other similar calls are made. A small test program `test` is included in the common directory, and is made with "`make test`". It takes as input a string representation of attributes, and

exercises the handlers. If this works for your new attribute, then it should be safe to recompile the DSA/DUA and try them.

If you only wish your DUAs to know about the new syntax, then add the call into your DUA's code, before loading the oidtables or calling `dsap_init`.

16.4.2 Modifying Previous Definitions

When developing DUAs, it is often required to handle standard attribute syntaxes in a different way to the predefined QUIPU handlers.

The procedure

```
set_attribute_syntax (sntx,enc,dec,parse,
                      print,cpy,cmp,sfree,
                      print_pe,approx,multiline)
short sntx;
IFP enc,dec,parse,print,cpy,cmp,sfree,approx;
char *  print_pe;
char multiline;
```

is used to modify an existing syntax handler. The parameters are exaclty the same as for the *add_attribute_syntax* except for the first *sntx* parameter, which should be the *short* value returned by the *add_attribute_syntax* call. If you do not know the value the procedure

```
short str2syntax (str)
char * str;
```

can be used to determine it, given the textual syntax name. The reverse mapping is preformed by

```
char * syntax2str (sntx)
short sntx;
```

Often in DUA development you do not want to alter all the handling routines, but just the attribute value printers. The routine

```
set_av_printer (sntx,print)
short sntx;
IFP print;
```

is used to alter the print routine for the given syntax. This is changed for *all* attributes using this syntax. If you only want to alter the handler for a specific attribute, then you should use

```
short modify_av_printer (at,sntx,print)
AttributeType at;
char * sntx;
IFP print;
```

The parameters are:

`at:` The attribute to alter the printer for,

`sntx:` The new string used to reference the syntax,

`print:` The print function.

Finally, there is a routine to alter the external process printer:

```
set_av_pe_print (sntx,print_pe)
short sntx;
char *  print_pe;
```

Chapter 17

The Procedural DUA

The *libdsap*(3n) library defines a set of procedure calls which correspond to each of the X.500 abstract operations. This chapter describes those procedure calls.

17.1 Procedure Model

Each operation is accessed via a procedure call of the same name as the X.500 operation, prefixed by "`ds_`". The procedure is supplied an argument structure, and returns either an error or result structure. For example the `read` operation is invoked by calling `ds_read (argument, error, result)`.

The return value of the procedures have the following common values, which indicated whether an error or result is returned:

`DS_OK`: Operation completed successfully; the result structure will have the corresponding result (if the operation generates results).

`DS_ERROR_LOCAL`: Error within the DUA module.

`DS_ERROR_CONNECT`: Failed to connect to a remote DSA.

`DS_ERROR_PROVIDER`: Other OSI provider error.

`DS_ERROR_REMOTE`: Remote error. Further details will be in the error parameter in the procedure call.

`DS_X500_ERROR`: This is the same as `DS_ERROR_REMOTE`.

These values defined in *quipu/ds_error.h*, which must be included in your program (there are other return values defined here, but these should only occur within the DSA, and should not be returned by a DUA call).

17.2 Common Parameters

All of the DAP operations described in Sections 17.5 to 17.11.4 have certain common parameters in their arguments and results. These structures are now described.

17.2.1 Arguments

The common arguments are represented by the following structure:

```
typedef struct common_args {
    ServiceControl        ca_servicecontrol;
    DN                    ca_requestor;
    struct op_progress    ca_progress;
    int                   ca_aliased_rdns;
#define CA_NO_ALIASDEREFERENCED -1
    struct security_parms * ca_security;
    struct signature    * ca_sig;
    struct extension    * ca_ext;
} CommonArgs;
```

The fields `ca_ext`, `ca_progress`, `ca_requestor` and `ca_aliased_rdns` are provided as they are defined within X.500. Neither the QUIPU DSA or DUA use these fields.

The field `ca_servicecontrol` is used to select the type of service the DSA should provide, the structure is given below

```
typedef struct svccontrol {
    int         svc_options;
#define SVC_OPT_PREFERCHAIN             0X001
#define SVC_OPT_CHAININGPROHIBIT        0X002
#define SVC_OPT_LOCALSCOPE              0X004
#define SVC_OPT_DONTUSECOPY             0X008
#define SVC_OPT_DONTDERFERERENCEALIAS   0X010
```

```
    char        svc_prio;
#define SVC_PRIO_LOW    0
#define SVC_PRIO_MED    1
#define SVC_PRIO_HIGH   2
    int         svc_timelimit;
#define SVC_NOTIMELIMIT -1
    int         svc_sizelimit;
#define SVC_NOSIZELIMIT -1
    int         svc_scopeofreferral;
#define SVC_REFSCOPE_NONE       -1
#define SVC_REFSCOPE_DMD        0
#define SVC_REFSCOPE_COUNTRY    1
} svccontrol, ServiceControl;
```

The values takes by each field within the ServiceControl structure are described in full in [CCI88].

17.2.2 Results

```
typedef struct common_results {
    DN      cr_requestor;
    char    cr_aliasdereferenced;
    struct security_parms  * cr_security;
    struct alg_id          * cr_alg;
    char                   * cr_tmp;
    int                      cr_len;
} common_results, CommonResults;
```

The filed `cr_aliasdereferenced` is set to `TRUE` if the base object of the operation was an alias, and was dereferenced.

The other fields are used by *pepsy* whilst encoding and decoding the structure.

17.3 Continuation References

"Continuation References" are returned by the directory when the operation asked for could not be fully completed, in which case the structure