7.5.4 Choice

A Choice is another example of a constructed type and so requires `T` value statement. A Choice also needs an integer field for the selection value to go. In encoding this is used to determine which type to choose and for decoding it is set to indicate which type was found. It is used just like pepy with value of 1 meaning the first type and 2 the second and so on. The selection field must be a field of the current structure and if it is left out will default to the *posy* generated value of "offset", which will generate syntax errors if this field is not defined when the corresponding tables field is compiled. There is an example of a choice in the previous Figure 7.3.

There is another "pepy-like" method of specifying which choice to take, which is illustrated in figure 7.4. The syntax allows a C expression to specify which choice to take when encoding and for decoding a C lvalue of an integer in which to store the choice present in the data. If the choice taken in decoding is not important or can be worked out from other information zero can be specified for the lvalue and the choice taken will not be recorded which is a bit more efficient. This is to support people changing *pepy* code and allows a way to those difficult bits which just can't be done by the first method. It is not as efficient as the first as it obviously requires some C code to be especially evaluated for the choice.

7.5.5 Default

Most defaults are handled transparently in *pepsy*. The user fills in what ever value required and *pepsy* will detect if that value is the default or not and encode it appropriately. Similarly on decoding it will decode into a normal value. The cases where it doesn't do this are constructed types. This has only been a problem so far with the default value being the empty sequence or set. At the moment the work around for this problem is to change the default to optional and have the data carried in another structure. To the default null value is carried by setting the pointer the structure to null. This is pretty much the way *posy* handles these situations. Note: you can not specify a default value for an Object Identifier at the moment, again use a work around based on OPTIONAL.

```
SearchResultData [[P struct ds_search_result *]]
        ::=
        CHOICE <D<(*parm)->srr_correlated>>
        <E<((parm->srr_correlated == FALSE) ? 2 : 1)>>
        {
        searchInfo
                SET [[ T struct ds_search_unit * $ srr_un.srr_unit ]]
.....
                },                                                          10
        uncorrelatedSearchInfo
                [0] SET OF [[ T struct ds_search_result * $ srr_un.srr_parts ]]
                    <<srr_next>> SearchResult [[p *]]
        }
```

Figure 7.4: Pepy-like selector in a choice

7.5.6 Optional

Optionals are handled in several ways. The original mannar used in *posy* which is a null pointer for those types specified by a pointer and a bit in an integer field called `optionals`. The first optional being specified by the bit 01, the 2nd by 02, the third by 04 and so on. The *pepsy* generated data structures still contain this field and #defines for the various bits if the ASN.1 type requires it, such as INTEGER or ENUMERATED type.

The next method for handling optionals is a generalisation of the *posy* scheme. The user may specify a bit out of an array of bytes (chars) which specifies whether or not the element is present. To use this technique you add after the OPTIONAL and before the `,` or terminating `}` the follow:

```
<< char_array $ Bit_Number>>
```

Where `char_array` is the character array field and `Bit_Number` is actual number of the bit used for this type, usually a #define symbol. Figure 7.2 shows some use of this form of OPTIONAL.

The third and final method of supporting optional is modeled on that used by pepy. It is the same as that used by the "pepy-like" choice selector which allows a C expression for encoding and an integer lvalue for decoding.

```
....
[30] ServiceControls [[p &parm->cma_common.ca_servicecontrol]]
                -- DEFAULT {} ,
               OPTIONAL <D<0>>
               <E< (
               (parm->cma_common.ca_servicecontrol.svc_options != 0) ||
               (parm->cma_common.ca_servicecontrol.svc_prio != SVC_PRIO_MED
) ||
               (parm->cma_common.ca_servicecontrol.svc_timelimit != SVC_NOT
IMELIMIT) ||                                                          10
               (parm->cma_common.ca_servicecontrol.svc_scopeofreferral != S
VC_REFSCOPE_NONE)
               ) >>,

....
```

Figure 7.5: Optional C-expression/lvalue

As in the choice case it is not as efficient as the previous two methods but is much more general. But for optionals the meaning of the integer is slightly different, non zero means present and zero means absent. Figure 7.5 shows an example of this optional. In this case the the user is not interested in being notified if it is present in the decoded and so specifies zero for the lvalue.

7.5.7 Encoding and Decoding Functions

Figure 7.6 shows the usage of these functions. By placing the `[[X fn ]]` after the type reference the handling for encoding or decoding can be passed to a user written function, called **fn**. The encoding is specified if **X** is an **E** and decoding is specified by an **D**. If only one of encoding or decoding is specified then the task left unspecified is done normally, see in the example the type E-ANY, which is encoded by calling the user function `get_pe()` and decoded by the value passing statement `[[ a * ]]`.

The definition of the encoding and decoding functions must follow a very precise calling and return value conventions. These are not the same as

```
E-ANY ::= ANY [[ a * ]] [[ E get_pe ]]

D-ANY ::= ANY [[ a * ]]  [[ D put_pe ]] [[ F fr_pe ]]

DE-ANY ::= ANY [[ a * ]]  [[ D put_pe ]] [[ E get_pe ]]

CodeOpt [[ P struct codedata * ]] ::= SEQUENCE {
    a [PRIVATE 5] BasicANY [[ p cd_a ]] OPTIONAL,
    b [35] E-ANY [[ p cd_b ]] OPTIONAL,
    c [45] EXPLICIT D-ANY [[ p cd_c ]] OPTIONAL <<cd_opt_set $ CD_C>>,
    d [1] DE-ANY [[ p cd_d ]] OPTIONAL <<cd_opt_set $ CD_D>>,
    e [15] IMPLICIT INTEGER [[ i cd_int ]] [[ E get_int ]] [[ D put_int ]]
            OPTIONAL << cd_opt_set $ CD_INT >>,
    f GraphicString [[ s cd_string ]] [[ E get_string ]][[ D put_string ]]
            OPTIONAL
}
```

Figure 7.6: Example of Encoding/Decoding functions

```
get_int(parm, ppe)
struct codedata *parm;
PE      *ppe;
{
    if (ppe == (PE *)0) {
        if (parm->cd_int == 1)  /* Default case */
            return (OK + 1);
        return (OK);
    }

    *ppe = int2prim(parm->cd_int);
    return (OK);
}
```

Figure 7.7: Definition of an Encoding function

those used by *pepy* generated functions which requires lots of parameters, but it shouldn't be too hard to convert *pepy* generated functions into this form. Figure 7.7 shows the definition of a simple but complete example of encoding function.

The encoding function is passed a pointer, **parm**, to the structure that contains the data at the level of the encoding function and the address of a PElement pointer, **ppe**. It expects the user to encode the data present some where in the data structure into a PElement and set the pointer to this PElement. If everything works it should return OK, if something fails it should return NOTOK, to abort the whole encoding.

The OPTIONAL and DEFAULT need to be detected by the function and it should return `OK + 1` and not encode anything. In the example it checks for the value 1 which is the default value and returns correctly otherwise encodes the value and returns `OK`. Note: the tag and class do not need to be set as pepsy will take care of getting that correct. This allows the same function to be used in several places, with different tags.

The decoding function also takes two pointers as arguements and returns `OK` on success and `NOTOK` on failure. The two pointers are different versions of the encoding function pointers, this time the first pointer is the address of the pointer to the decoding function and a pointer to the PElement containing

```
put_int(parm, pe)
struct codedata **parm;
PE      pe;
{
   if (pe)
      (*parm)->cd_int = prim2num(pe);
   else
      (*parm)->cd_int = 1;    /* Default */
   return (OK);
}                                                              10
```

Figure 7.8: Definition of a Decoding function

the encoded data. The function is expected to construct the decoded data type which may include allocating space for it and setting pointers in the `*parm` data structure to the allocated space. If the PElement pointer is null (there is no encoded data) it means the type is absent and it is up to the function to handle it. This would be the case if it were DEFAULT or OPTIONAL element.

7.5.8 Pepsy C Action Statements

7.5.9 External Type References

Part IV

Recipes

Chapter 8

Run-Time Environment

The *librosap*(3n) library implements the run-time environment for the stubs generated by the *rosy*(1) program.

The routines described herein provide a more structured interface to the remote operations service (ROS). They are used in the boilerplate routines which are described in the following chapters. Hence, the more hasty reader may wish to skip this chapter on first reading of this volume.

8.1 Notice

Please read the following important message.

> **NOTE:** When using the *librosy*(3n) library, no calls should be made to the *librosap*(3n) library. (That is, the `RoXXX` routines should not be directly referenced.)
> There are three exceptions: `RoSetService` and its related arguments (e.g., `RoPService`), `RoSelectMask`, and `RoErrString`.

8.2 Conventions

All of the routines in the *librosy*(3n) library are integer-valued. Unless otherwise indicated, they return the manifest constant `OK` on success, or `NOTOK` otherwise.

The routines all work on an established association, which is referenced by an association-descriptor. This is usually the first parameter given to any of the routines in the *librosy*(3n) library. Further, the last parameter is usually a pointer to a `RoSAPindication` structure. If a call to one of these routines fails, then the structure is updated. Section 3.3 of *Volume One* describes this structure and how to interpret error conditions and their severity.

8.2.1 Interface from ROSY

For a given module, e.g., `MODULE`, *rosy* will define two tables:

```
struct RyOperation table_MODULE_Operations[];

struct RyError table_MODULE_Errors[];
```

The `RyOperation` structure is used to define the compiled parts of an operation:

```
struct RyOperation {
    char   *ryo_name;
    int     ryo_op;

    IFP     ryo_arg_encode;
    IFP     ryo_arg_decode;
    IFP     ryo_arg_free;

    int     ryo_result;
    IFP     ryo_res_encode;
    IFP     ryo_res_decode;
    IFP     ryo_res_free;

    struct RyError **ryo_errors;
};
```

The elements of this structure are:

`ryo_name/ryo_op:` the operation name and code;

`ryo_arg_encode/ryo_arg_decode:` the address of the routines to encode or decode (respectively) the operation's argument from a *C* structure to its corresponding abstract syntax (present only if the operation takes an argument);