```
}
```

the control `<< 2 >>` would choose the second type in a choice, which is not necessarily the element tagged with `[2]`, as in this case. The statement always precedes the opening brace of the `CHOICE`.

- `SEQUENCE␣OF` and `SET␣OF` — to indicate the number of elements require. This construct has the syntax of a for loop in C. Elements are added to the `SEQUENCE/SET` while the second expression yields `TRUE`. The sequence

```
SeqDef ::=
    SET OF << count = 0; count < 10; count ++ >>
        MatchingAccessories
```

  would produce ten elements in the set. The statement follows the `OF` and any optional actions.

- `OPTIONAL` — to decide whether the optional element should be included. This statement follows the `OPTIONAL` keyword. It is a simple boolean expression. If the value returned is `TRUE` then the element is included.

- `DEFAULT` — to decide whether the optional element should be included. This statement follows the `DEFAULT` keyword/value pair. It is a simple boolean expression. If the value returned is `TRUE` then the element is included.

  Arguably, this should be unnecessary — *pepy* should look at the value immediately following `DEFAULT` to decide if the element need be encoded.

The example in Figure 6.4 should make this a little clearer.

Value Passing Statements

A number of extensions are available to assist in passing values between the generated routines. The generated routines pass these values between rules as parameters to each routine. This can obviate the need for much of the *C* code associated with each type.

```
ENCODER build

Example ::=
        SEQUENCE {
           -- select maybe element if present
           Maybe
           OPTIONAL <<maybe == PRESENT>>,

           SEQUENCE OF
                -- build elements                                    10
                %{ construct_elements(); %}
                -- now produce them
                <<init_elements(); more_elements();>>
                BizarreEvents,

           CHOICE
           %{ choose = leap_year() ? 2 : 1; %}
           <<choose>>
           {
                [0] MyProposal,                                     20
                [1] YourProposal
           }

           INTEGER
           [[i defvalue]]
           DEFAULT 4 <<defvalue != 4>>
        }
```

Figure 6.4: Examples of CONTROL actions

To allow values to be passed to and obtained from these parameters, value tokens can be associated with the type definitions. These tokens are incorporated in the lexemes `[[` and `]]`. The tokens come in two types. The first is a key letter followed by a expression. The second is the same with an additional parameter to specify the length of the first parameter. These formats are shown below.

- `[[b expression]]` — a boolean. For encoding, this is the value that the presentation element will take. For decoding and printing, the value of the presentation element is assigned to this variable.

- `[[i expression]]` — an integer. For encoding, this is the value that the presentation element will take. For decoding and printing, the value of the presentation element is assigned to this variable.

- `[[s expression]]` — a string. For encoding, this is the value that the presentation element will take, a null-terminated string. For decoding and printing, the string value of the presentation element is assigned to this variable; the user is responsible for freeing the storage assigned with `free`.

- `[[O expression]]` — an object identifier. For encoding, this is the value that the presentation element will take, an `OID` value (see Section 5.4.6 on page 135 in *Volume One*). For decoding and printing, the `OID` value of the presentation element is assigned to this variable; the user is responsible for freeing the storage assigned with `oid_free`.

- `[[p expression]]` — a user supplied parameter, `parm`. This parameter is of type `PEPYPARM`, which can be `#define`'d by the user to a suitable value. If not defined, it defaults to a `char␣*`. Most commonly this type is defined to be a union of all parameter types that need to be passed (but see the next item).

- `[[P type]]` — a *C* type which defines the type of the parameter for this definition. If present, this construct must occur directly after the definition name.

- `[[o expression1 $ expression2]]` — a byte pointer/length pair. For encoding, these indicate the byte value and length of the presentation element, which needn't be null-terminated. For decoding and

printing, the byte value and length are assigned to these variables; the user is responsible for freeing the storage assigned with `free`.

- `[[q expression]]` — a `qbuf`. For encoding, this is a pointer to a `qbuf` (described in Section 5.4.4 in *Volume One*) which is used to build the presentation element. If the `qbuf` contains only one element in the linked list, then a primitive element is built, otherwise a constructed type is built. which needn't be null-terminated. For decoding and printing, this value is undefined.

- `[[x expression1 $ expression2]]` — a bit pointer/length pair. For encoding, the value of the presentation element is derived by supplying `expression1` and `expression2` as the first two arguments to `strb2bitstr` (described in Section 5.4.5 in *Volume One*). For decoding and printing, the bitstring value and length are assigned to these variables; the user is responsible for freeing the storage assigned with `free`.

- `[[a expression]]` — a presentation element. For encoding, this is the value that the presentation element will take. For decoding and printing, the presentation element is assigned to this variable.

- `[[r expression]]` — a real number. For encoding this is a floating point number that will be encoded. For decoding this represents a floating point number variable in double format.

The example in Figure 6.5 should demonstrate these features.

```
SalaryDefs DEFINITIONS ::=

%{
#include <stdio.h>

char   *myname;

static struct salary_record {
   char   *name;
   int     salary;                                          10
}         salary;

#define PEPYPARM struct salary_record *


main (argc, argv)
int     argc;
char  **argv;
{
   PE       pe;                                             20

   myname = argv[0];

   salary.name = argv[1];
   salary.salary = atoi (argv[2]);
   build_SalaryDefs_Salary (&pe, 1, 0, NULLCP, &salary);

   salary.name = NULL;
   salary.salary = 0;
   unbuild_SalaryDefs_Salary (pe, 1, NULLIP, NULLVP, &salary);  30

   exit (0);
}
%}

...
```

Figure 6.5: Examples of Parameters in Pepy

```
...

BEGIN

SECTIONS build unbuild none

Salary ::=
        SEQUENCE {
            name
                PrintableString [[s parm -> name]]                    10
                %{ printf("name %s ", parm -> name); %},

            salary
                TheSalary [[i parm -> salary]]
                %{ printf("salary %d\n", parm -> salary); %}
}

TheSalary ::=
        INTEGER
                                                                      20
END
```

Figure 6.5: Examples of Parameters in Pepy (continued)

6.3 Known Deficiencies

The *pepy* program started as a prototype. Although the approach is promising, some of the rough edges remain.

6.3.1 ASN.1 Syntax

The ASN.1 syntax is followed exactly except that:

- No *macro* syntax is recognized.

6.3.2 ASN.1 Semantics

`COMPONENTS OF` and `CHOICE` types are often "pulled up" into the type which references them. This can cause anomalous behavior when things like value passing statements (e.g., `[[p expr]]`) are present in the definition of a `COMPONENTS OF` or `CHOICE` type which is pulled into another type definition.

6.4 PEPY Environment

A program generated by *pepy* expects certain things.

6.4.1 Starting Things Off

To invoke a decoding or printing routine:

```
PREFIX_MODULE_TYPE (pe, 1, len, buffer, parm)
```

where –

`PREFIX_MODULE_TYPE:` indicates that the type `TYPE` in module `MODULE` should be parsed – the prefix `PREFIX` is supplied from the `DECODER` or `PRINTER` directive (or the corresponding prefix in the `SECTIONS` directive).

`pe:` is the presentation-element that is to be examined.

`1:` is a magic argument.

`len:` is the integer parameter supplied for routines that decode values into integer parameters, or have a length associated with them. The address of an integer can be placed here if the element being parsed is a primitive type, otherwise use `NULLIP`.

`buffer:` is the string parameter for routines that decode values into string objects. The address of a character pointer can be placed here if the element being parsed is a primitive type, otherwise use `NULLVP`.

`parm:` is the user supplied parameter. This should be a variable of type `PEPYPARM` if supplied, otherwise use `NULLCP`.

To invoke an encoding routine, the call is similar:

```
PREFIX_MODULE_TYPE (&pe, 1, len, buffer, parm)
```

the values are the same as above, except that the first parameter is a reference higher and the third and fourth parameters are a reference lower, hence –

`PREFIX_MODULE_TYPE:` indicates that the type `TYPE` in module `MODULE` should be parsed – the prefix `PREFIX` is supplied from the `ENCODER` directive (or the corresponding prefix in the `SECTIONS` directive).

`pe:` is a pointer to the presentation-element that is to be constructed.

`1:` is a magic argument.

`len:` is the integer parameter supplied for routines that encode booleans or integers, or have a length associated with them. If not present, use `NULL`.

`buffer:` is the string parameter for routines that encode string values. If not present, use `NULLCP`.

`parm:` is the user supplied parameter. This should be a variable of type `PEPYPARM` if supplied, otherwise use `NULLCP`.

These routines return the manifest constant `NOTOK` on error, or `OK` on success.

6.4.2 Diagnostic Output

When the program generated by *pepy* detects an error, it calls the routine `advise` with some diagnostic information. This routine, which should be declared `void` is defined by the user. The routine takes a variable number of arguments, similar to a *printf*(3s) routine, with the exception that the first argument is a string for *perror*(3), the second argument is the format string, and any following arguments are referenced by the format string. Consult Figure 6.11 on page 85 for an example of the `advise` procedure.

Alternately, the standard routine, `PY_advise` can be used. It simply records the information to:

```
char   PY_pepy[BUFSIZ];
```

for easy access. Note that the routines in the *libpepy*(3) library are generated to use the `PY_advise` routine. This routine is also encouraged for use when building *rosy* encoder/decoder routines.

6.4.3 Debug Output

If a program generated by *pepy* is compiled with the `-DEBUG` option, then prior to fiddling with each instance of a type, the routine `testdebug` is called.

```
int     testdebug (pe, s)
PE      pe;
char   *s;
```

The parameters to this procedure are:

- the presentation element being examined; and,
- a short null-terminated diagnostic string describing the instance of the type.

By convention, if the program was compiled with `-DDEBUG`, then the `testdebug` routine (supplied by the user) should examine the `$PEPYDEBUG` environment variable. If this variable is set to a non-zero decimal number, then it should print out debugging information. Consult Figure 6.11 on page 85 for an example of the `testdebug` procedure.

In addition, the static character pointer `pepyid` contains information on the version of *pepy* which generated the program.

6.5 Pretty-printers

To facilitate the automatic generation of pretty-printers, the routine `PY_pp` is included in the *libpepy*(3) library. Given an invocation from a simple main routine, `PY_pp` will parse invocation arguments, build presentation elements (from either a presentation stream or presentation list), and then invoke the pretty-printer routine for each presentation element. By simply invoking pepy with the '`-S␣PRINT`' option. the pretty-printer routine can be generated automatically. Figure 6.6 shows the required initialization code for the pretty-printer.

```
/* prpp.c - Personnel Record prety-printer */

#include <stdio.h>

int     print_PERSONNEL_PersonnelRecord ();


main (argc, argv, envp)
int     argc;
char  **argv,                                                    10
     **envp;
{
   exit (PY_pp (argc, argv, envp, print_PERSONNEL_PersonnelRecord));
}
```

Figure 6.6: Simpler Pretty-Printer initialization

6.6 Compiling and Loading

Programs generated by *pepy* automatically include `<isode/psap.h>`. Typically, these programs should also be loaded with `-lisode`, although only the libraries `-lpepy`, `-lpsap`, and `-licompat` are referenced.

6.6.1 External Modules

When an external module, `MODULE`, is referenced, *pepy* reads a file which is named `MODULE.ph` for various information. The *pepy* program will consult the environment variable `$PEPYPATH` as a search-list of directories to search to find this file. The components in the path are separated by a colon (':'). If this environment variable is not set, then *pepy* consults the user's current directory followed by a system-wide area defined at compile-time.

Similarly, in addition to producing a `.c` file, *pepy* will also generate a `.ph` file containing information describing the module.

6.6.2 Options

The *pepy* program has many options to modify its behavior: