Normally, *rosy* prints the name of each operation, error, or type as it works. The '`-s`' switch disables this behavior.

The '`-pepsy`' switch tells *rosy* that you will be using *pepsy* rather than *posy*. The *pepsy* program is dscribed in Chapter 7 starting on page 88.

4.4.2 Makefiles

By convention, input files to *rosy* have the extension `.ry`.

Consider the file `module.ry` which defines the module `MODULE`. The name of the abstract syntax module produced by *rosy* is normally derived from the name of the input file, using an extension of `.py`. The name of the *C* language stub files is derived from the name of the module defined in the input file. Hence

```
rosy module.ry
```

will produce:

`module.py:` the abstract syntax module;

`MODULE-ops.h:` operation definitions and stubs;

`MODULE-ops.c:` data structures for *librosy*(3n); and,

`MODULE-stubs.c:` stub definitions for *lint*(1)

Intuitively, one would suspect that the the rules for *make*(1) should take advantage of the `.ry` and `.py` suffix relationship for a rule like:

```
.ry.py:;          rosy $(RYFLAGS) $<
```

For reasons explained later on (in Section 5.4.2 on page 53), it is advantageous to use a more complicated set of rules:

```
MODULE-asn.py:    module.ry
                  rosy $(RYFLAGS) -o $@ module.ry
MODULE-ops.c:     module.ry
MODULE-ops.h:     module.ry
MODULE-stubs.c:   module.ry
```

Usually `RYFLAGS` is empty.

In our particular discipline, we will generate two sets of files: one for the invoker and the other for the performer.

```
MODULE-Rops.o:    MODULE-ops.c MODULE-ops.h
                  $(CC) $(CFLAGS) -DPERFORMER -c MODULE-ops.c
                  mv MODULE-ops.o $@

MODULE-Iops.o:    MODULE-ops.c MODULE-ops.h
                  $(CC) $(CFLAGS) -DINVOKER -c MODULE-ops.c
                  mv MODULE-ops.o $@
```

4.4.3 Grinding

Figure 6.7 on page 79 of *Volume One* contains an entry for the *vgrindefs*(5) file, which may be used by the *vgrind*(1) or *tgrind*(1) facility.

Chapter 5

Pepy Optional Structure-Generator

The *posy*(1) program is a compiler for specifications of abstract syntaxes.

For those interested in trivia, *posy* stands for Pepy Optional Structure-generator (YACC-based). The *pepy*(1) program is an ASN.1 compiler, and the *yacc*(1) program, of course, is a compiler-compiler.

5.1 Abstract Syntax Module

The *posy* program reads a description of an *abstract syntax module* and produces the corresponding *C* structures along with an augmented abstract syntax module.

The syntax of the input file corresponds almost exactly to the notation described in [ISO87, CCI88c]. (But see Section 5.3 on page 52 for the details.) As the ASN.1 language is rich in its notation, the rules are not presented herein, though Section 6.2.1 on page 60 of *Volume One* presents an incomplete treatment of the less substantive aspects of the language.

An example of a complete abstract syntax module can be found in Figure 5.1 starting on page 42. This corresponds to the output produced by *rosy*(1) when given the remote operations module in Figure 4.1 as input.

For our purposes, the input file is relatively unimportant as it is usually generated by the *rosy* program. Hence, the output from *posy* is where this chapter focuses.

```
-- automatically generated by rosy 3.1 #15 (killer-rat), do not edit!

IMAGE DEFINITIONS ::=

BEGIN

Initialize-INV ::=
        SEQUENCE {
           proposed[0]
                IMPLICIT Version                                    10
                DEFAULT { version-1 }
        }

Initialize-RSP ::=
        SEQUENCE {
           negotiated[0]
                IMPLICIT Version
                OPTIONAL
        }
                                                                    20
VersionsSupported ::=
        SEQUENCE {
           supported[0]
                IMPLICIT Version
        }

Version ::=
        BIT STRING {
           version-1(0)
        }                                                           30

ReadImage-INV ::=
        SEQUENCE {
                SEQUENCE OF
                   CHOICE {
                        file-name[0]
                           IMPLICIT Filename,
```

```
                        mail-address[1]
                            IMPLICIT Mail-Address
                    },

                SEQUENCE OF
                    Format
        }

ReadImage-RSP ::=
        SEQUENCE {
                CHOICE {
                    x-windows
                        X-Windows-Bitmap
                }
        }

FormatsSupported ::=
        SEQUENCE {
            supported[0]
                IMPLICIT Format
        }

Format ::=
        BIT STRING {
            x-windows-bitmap(0)
        }

Filename ::=
    [APPLICATION 1]
        IMPLICIT GraphicString

Mail-Address ::=
    [APPLICATION 2]
        IMPLICIT SEQUENCE {
            local[0]
                IMPLICIT GraphicString,

            domain[1]
                IMPLICIT GraphicString,
```

```
        options[2]
                IMPLICIT BIT STRING {                                    80
                    default-local(0),
                    default-host(1)
                }
                DEFAULT { default-local,  default-host }
    }

X-Windows-Bitmap ::=
  [APPLICATION 100]
        IMPLICIT SET {
            width[0]                                                     90
                IMPLICIT INTEGER,

            height[1]
                IMPLICIT INTEGER,

            hot-spot-x[2]
                IMPLICIT INTEGER
                DEFAULT -1,

            hot-spot-y[3]                                               100
                IMPLICIT INTEGER
                DEFAULT -1,

            data[4]
                IMPLICIT OCTET STRING
        }

END
```

Figure 5.1: Example of an Abstract Syntax Module

5.2 POSY Environment

The *posy* program will produce two files after reading its abstract syntax module.

5.2.1 C Language Structures

For each type in the abstract syntax module, *posy* will define a corresponding *C* language structure (`struct`).

The rules that *posy* uses to generate the structures are fairly simple:

`BOOLEAN:` represented as a `char`.

`INTEGER:` represented as an `int`. If the `INTEGER` type has a list of tagged values, then for each value:

```
#define int_MODULE_TYPE_TAG value
```

as in:

```
#define int_IMAGE_IntList_zero 0
```

`BIT STRING:` represented as a `struct PElement` (these are described in Section 5.4.5 of *Volume One*). If the `BIT STRING` type has a list of tagged bits, then for each bit:

```
#define bit_MODULE_TYPE_TAG value
```

as in:

```
#define bit_IMAGE_Version_version__1 0
```

In addition, one other definition is present:

```
#define bits_MODULE_TYPE string
```

as in:

```
#define bits_IMAGE_Version "\020\01version-1"
```

`OCTET STRING:` represented as a `struct qbuf` (these are described in Section 5.4.4 of *Volume One* and Section 4.3.2 of *Volume Two*).

`REAL:` represented as a `double`.

`NULL:` represented as a `char`.

`SEQUENCE/SET/ANY:` represented as a `struct PElement` (which are described throughout Chapter 5 of *Volume One*). This format is used for those `SEQUENCE`s or `SET`s which contain no elements.

`SEQUENCE OF/SET OF:` by default, represented as a `struct` with two elements, a linked-list. The first element is the type for set or sequence, the second element is a pointer to another instance of the structure. For example:

```
SEQUENCE OF
    Format
```

might be translated as:

```
struct element_IMAGE_4 {
    struct type_IMAGE_Format *Format;

    struct element_IMAGE_4 *next;
};
```

By using the '`-h2`' option (described below), an array structure can be generated instead.

`SEQUENCE/SET:` represented as a simple `struct` with an element for each member. For example:

```
Mail-Address ::= SEQUENCE {
    local[0]
        IMPLICIT GraphicString,

    domain[1]
        IMPLICIT GraphicString,

    options[2]
        IMPLICIT BITSTRING
}
```

might be translated as:

```
struct type_IMAGE_Mail__Address {
    struct type_UNIV_GraphicString *local;

    struct type_UNIV_GraphicString *domain;

    PE      options;
};
```

`CHOICE`: represented as a `struct` containing an `int` and a `union`.

```
CHOICE {
    file-name[0]
        IMPLICIT Filename,

    mail-address[1]
        IMPLICIT Mail-Address
}
```

might be translated as:

```
struct choice_IMAGE_0 {
  int     offset;
#define choice_IMAGE_0_file__name       1
#define choice_IMAGE_0_mail__address    2

  union {
     struct type_IMAGE_Filename *file__name;

     struct type_IMAGE_Mail__Address *mail__address;
  }   un;
};
```

`OBJECT IDENTIFIER`: represented as a `struct OIDentifier` (these are described in Section 5.4.6 of *Volume One*).

`MODULE.TYPE`: represented as a pointer to a `struct type_MODULE_TYPE`.

Although these rules might seem complicated, in point of fact they are actually very straight-forward and regular. The easiest way to gain familiarity with them is to run some abstract syntax modules through *posy* and

then view the resulting definitions. Figure 5.2 starting on page 51 presents the *C* language structures corresponding to the abstract syntax module in Figure 5.1.

Controlling the names that POSY generates

One of the least friendly aspects of *posy* is the algorithm it uses for choosing the names of *C* language structures being generated. The rules for choosing some names are fixed, i.e., the top-level name generated for an ASN.1 type cannot be changed. However, the names used for the parts of the `struct`s generated by *posy* can be controlled — to an extent using either (or both) of two techniques.

The first method takes advantage of the observation that *posy* uses commentary tags in the ASN.1 definition, whenever possible, as variable names. To revisit an earlier example:

```
Mail-Address ::= SEQUENCE {
    local[0]
        IMPLICIT GraphicString,

    domain[1]
        IMPLICIT GraphicString,

    options[2]
        IMPLICIT BITSTRING
}
```

might be translated as:

```
struct type_IMAGE_Mail__Address {
    struct type_UNIV_GraphicString *local;

    struct type_UNIV_GraphicString *domain;

    PE      options;
};
```

In each case, the names of the variables (e.g., `local`) were taken from the commentary tag. Note that if no tag is present, *posy* must choose a name in a different fashion. Hence, the first level of control is to simply add short,