```
    int     hot_spot_y;
                                                                    80
    struct qbuf *data;
};
#endif
```

Figure 5.3: Example of C Language Structures (with -h0)

```
...

struct type_IMAGE_ReadImage_INV {
   struct element_IMAGE_1 {
          int nelem;
          struct choice_IMAGE_0 {
             int     offset;
#define choice_IMAGE_0_file_name          1
#define choice_IMAGE_0_mail_address       2
                                                                        10
             union {
                   struct type_IMAGE_Filename *file_name;

                   struct type_IMAGE_Mail_Address *mail_address;
             }      un;
          } *element_IMAGE_2[1];
   } *element_IMAGE_0;

   struct element_IMAGE_4 {
          int nelem;                                                    20
          struct type_IMAGE_Format *Format[1];
   } *element_IMAGE_3;
};
int       free_IMAGE_ReadImage_INV ();

...
```

Figure 5.4: Example of C Language Structures (with -h2)

Chapter 6

Compiling Data-Structures

The *pepy*(1) program is a compiler for the presentation elements discussed in Chapter 5 in *Volume One*. Although experimental in previous releases, the *pepy* program has now stabilized.

For those interested in trivia, *pepy* stands for Presentation Element Parser (YACC-based). The *yacc*(1) program, of course, is a compiler-compiler.[1]

6.1 Warning

Please read the following important message.

> **NOTE:** Users of *The Cookbook* emply the *posy*(1) program program as the front-end to *pepy*. Hence, this chapter may be safely skipped. It is included only for those misguided individuals who do not *posy*. This behavior is not recommended.

6.2 Syntax Rules

The *pepy* program reads a description of an *abstract syntax module* and produces a *C* program which recognizes the objects described in the module, or generates those objects, or prints those objects. The syntax of the input

[1] This abbreviation is now strictly incorrect, as the *pepy* program can now generate presentation elements as well as parse them.

file corresponds almost exactly to the Abstract Syntax Notation (ASN.1) discussed in Section 6.8 on page 87.

6.2.1 ASN.1 Notation

To summarize the ASN.1 syntax:

- Comments start with two adjacent dashes ('`--`') and are terminated by either new-line or another two adjacent dashes, whichever comes first.
- Whitespace between tokens is ignored.
- Keywords are entirely in uppercase.
- The names of rules (types and values) start with an uppercase character and may use either upper- or lowercase characters afterwards.
- The names of context-specific tags start with a lowercase character and may use either upper- or lowercase characters afterwards.
- A name is an alphabetic character followed by zero or more alphanumerics or dashes (except '`--`', of course).
- Literal strings are surrounded by double-quotes ('`"`').
- Numbers take on one of two forms:
  - a string of digits, optionally preceded by a minus sign, which is interpreted in decimal; and,
  - a string surrounded by single-quotes ('`'`') and immediately followed by one of `B`, `b`, `H`, or `h`, which is interpreted as either a binary (`B` or `b`) or hexadecimal (`H` or `h`) number.

An example of a specification using ASN.1 can be found in Figure 6.1.

6.2.2 ASN.1 Extensions

The *pepy* program recognizes several extensions to the ASN.1 syntax: *compiler directives*, *action statements*, *control statements*, and, *value passing statements*.

```
PERSONNEL DEFINITIONS ::=

BEGIN

PersonnelRecord ::=
        [APPLICATION 0]
          IMPLICIT SET {
                Name,

            title[0]                                                    10
                IA5String,

                EmployeeNumber,

            dateOfHire[1]
                Date,

            nameOfSpouse[2]
                Name,
                                                                        20
            [3]
                IMPLICIT SEQUENCE OF ChildInformation DEFAULT {}
          }

ChildInformation ::=
        SET {
                Name,

          dateofBirth[0]
                Date                                                    30
        }

Name ::=
        [APPLICATION 1]
          IMPLICIT SEQUENCE {
            givenName
                IA5String,

            initial
                IA5String,                                              40

            familyName
                IA5String
          }

EmployeeNumber ::=
        [APPLICATION 2]
          IMPLICIT INTEGER

Date ::=                                                                50
        [APPLICATION 3]
          IMPLICIT IA5String -- YYYYMMDD --

END
```

Figure 6.1: Example of a specification using ASN.1

Compiler Directives

The compiler directives control the type of *C* routines that are produced. The *pepy* program can generate routines to parse, create, and print presentation elements. The directives may occur after the `BEGIN` keyword; and anywhere an `END` keyword may occur. Each occurrence specifies the type of *C* routine to generate for successive type definitions and applies until the next directive.

The directives are:

`ENCODER` **build** — directs *pepy* to create routines for the construction of presentation elements;

`DECODER` **parse** — directs *pepy* to create routines to parse presentation elements;

`PRINTER` **print** — directs *pepy* to create routines which print the contents of presentation elements; or,

`SECTIONS` **build parse print** — directs *pepy* to create multiple forms of above.

The `build`, `parse`, and `print` strings are used to construct the name of the routine and are described later.

With the `SECTIONS` directive, the value `none` may be used as a null-valued place-holder. For example:

```
SECTIONS none undo pp
```

indicates that *pepy* should generate decoding and printing routines (using prefixes `undo` and `pp` respectively), and that no encoding routines should be generated.

There is also another directive, which must occur immediately *before* the `BEGIN` keyword:

`PREFIXES` **build parse print** — tells *pepy* what `SECTIONS` were used in other modules.

Action Statements

The lexemes `%{` and `}%` surround *C* code which the resulting program should execute at the appropriate time. The code may be of arbitrary length, spanning many lines in the input file.

Actions may occur in several places in the input file. Depending on the position of the actions, slightly different semantics are associated with the code. There are four types of actions: *verbatim actions*, *type actions*, *tag actions*, and, *declaration actions*.

The first type of actions are *verbatim* actions which occur in two places in the input file:

- immediately before the `BEGIN` keyword; and,
- after the `END` keyword in the input file.

These actions are copied strictly verbatim, being intended for definitions, subroutines, and the like.

The second type of actions are *type* actions which are placed after a type or sub-type definition, and after each element in a "named number". In the former case, please note that the action must be placed before the `OPTIONAL` and `DEFAULT` lexemes. During the copy, the lexeme `$$` is expanded to represent the object currently being examined. The exact expansion depends upon the element being processed and what sort of processing is being done. In decoding and printing routines, `$$` will expand to a `PE` in actions before the element is recognized and to the actual data after the processing. In the case of encoding routines, the opposite is true. The precise nature of the data is shown below.

Type	C declaration
`BOOLEAN`	`int`
`INTEGER`	`int`
`BITSTRING`	`PE` (a bitvector)
`OCTETSTRING`	`char *` (length is `$$_len`)
`REAL`	`double`
any `SEQUENCE`	`PE` (a sequence)
any `SET`	`PE` (a set)
`OBJECT␣IDENTIFIER`	`OID`

Consider the type definition in Figure 6.2:

- `action1` will be executed only if the object being recognized is a `document`;
- `action2` will be executed only if the object being recognized is a `paragraph`; and,
- `action3` will always be executed afterwards.

The third type of actions are *tag* actions which are placed after the tag for a tagged type or after the name for a named type. Consider the type definition in Figure 6.3:

```
LogicalObjectType ::=
        INTEGER {
                document(0)
                %{ action1 %},

                paragraph(1)
                %{ action2 %},
        }
        %{ action3 %}
```

Figure 6.2: Example of a TYPE action

```
ProtocolElement ::=
        CHOICE {
           textUnit[8]
                IMPLICIT TextUnit
                %{ action1 %},

           specificLogicalDescriptor[5]
                %{ action2 %}
                IMPLICIT LogicalDescriptor
        }
```

Figure 6.3: Example of a TAG action

- `action1` is executed after `textUnit` has been fully processed as a `TextUnit`; but,
- `action2` is executed after `specificLogicalDescriptor` has been recognized, but before it has been processed as a `LogicalDescriptor`. This type has both a name (`specificLogicalDescriptor`) and a tag (`[5]`). Although two actions are present, one only action is honored, and this action should be specified after the tag.

A tag action may also be specified for `SET␣{`, `SEQUENCE␣{` and `CHOICE␣{` – even when they are not tagged. The action is placed before the left brace, and is executed when the `SET`, `SEQUENCE`, or `CHOICE` is recognized but before

any of its members are processed.

A tag action may additionally be specified directly after the "::=" lexeme, even when not tagged. This has the same effect as the tagged rule. This is the only way to provide a pre-action for untagged single rules.

Finally, a tag action may occur after the phrase SET␣OF or SEQUENCE␣OF. This action is executed before processing each instance of the base type of the SEQUENCE or SET. This is useful for allocating memory for each of the member elements.

The final type of actions are declaration actions. These are designed to allow the declaration of variables specific to a generated function. Such actions occur between the type name and the "::=" lexeme. The statements placed here are appended to the declaration section of the generated routine. Only variable definitions should be placed here, as in:

```
X400 %{ char cp[100]; %} ::=
    SET {
        X4001984 %{ strcpy(cp, "1984"); %},
        X4001988 %{ strcat(cp, " 1988"); %} OPTIONAL,
        X4001992 %{ strcat(cp, " 1992"); %} OPTIONAL
    }
    %{ puts(cp); %}
```

Control Statements

When constructing routines for encoding presentation elements, additional constructs are required at certain points to control the choices available. The control constructs are specified between the lexemes << and >>. The control itself consists of an appropriate *C* statement which takes a different form for each of the constructs. The affected types are listed below.

- CHOICE — to decide which of the selection to choose. The statement consists of an expression of integer value that chooses the *n*'th value. For example in

```
NewDef ::=
    CHOICE << 2 >> {
        [0] FirstElement,
        [1] SecondElement,
        [2] ThirdElement
```