The PXP user’s guide

Gerd Stolpmann

The PXP user's guide
by Gerd Stolpmann

PXP is a validating parser for XML-1.0 which has been written entirely in Objective Caml.

Download PXP: The free PXP library can be downloaded at http://www.ocaml-programming.de/packages/. This user's guide is included. Newest releases of PXP will be announced in The OCaml Link Database (http://www.npc.de/ocaml/linkdb/).

License

Table of Contents

List of Figures

I. User’s guide

Chapter 1. What is XML?

1.1. Introduction

XML (short for *Extensible Markup Language*) generalizes the idea that text documents are typically structured in sections, sub-sections, paragraphs, and so on. The format of the document is not fixed (as, for example, in HTML), but can be declared by a so-called DTD (document type definition). The DTD describes only the rules how the document can be structured, but not how the document can be processed. For example, if you want to publish a book that uses XML markup, you will need a processor that converts the XML file into a printable format such as Postscript. On the one hand, the structure of XML documents is configurable; on the other hand, there is no longer a canonical interpretation of the elements of the document; for example one XML DTD might want that paragraphes are delimited by `para` tags, and another DTD expects `p` tags for the same purpose. As a result, for every DTD a new processor is required.

Although XML can be used to express structured text documents it is not limited to this kind of application. For example, XML can also be used to exchange structured data over a network, or to simply store structured data in files. Note that XML documents cannot contain arbitrary binary data because some characters are forbidden; for some applications you need to encode binary data as text (e.g. the base 64 encoding).

1.1.1. The "hello world" example

The following example shows a very simple DTD, and a corresponding document instance. The document is structured such that it consists of sections, and that sections consist of paragraphs, and that paragraphs contain plain text:

```
<!ELEMENT document (section)+>
<!ELEMENT section (paragraph)+>
<!ELEMENT paragraph (#PCDATA)>
```

The following document is an instance of this DTD:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE document SYSTEM "simple.dtd">
<document>
  <section>
    <paragraph>This is a paragraph of the first section.</paragraph>
    <paragraph>This is another paragraph of the first section.</paragraph>
  </section>
```

```
  <section>
    <paragraph>This is the only paragraph of the second section.</paragraph>
  </section>
</document>
```

As in HTML (and, of course, in grand-father SGML), the "pieces" of the document are delimited by element braces, i.e. such a piece begins with `<name-of-the-type-of-the-piece>` and ends with `</name-of-the-type-of-the-piece>`, and the pieces are called *elements*. Unlike HTML and SGML, both start tags and end tags (i.e. the delimiters written in angle brackets) can never be left out. For example, HTML calls the paragraphs simply `p`, and because paragraphs never contain paragraphs, a sequence of several paragraphs can be written as:

```
<p>First paragraph
<p>Second paragraph
```

This is not possible in XML; continuing our example above we must always write

```
<paragraph>First paragraph</paragraph>
<paragraph>Second paragraph</paragraph>
```

The rationale behind that is to (1) simplify the development of XML parsers (you need not convert the DTD into a deterministic finite automaton which is required to detect omitted tags), and to (2) make it possible to parse the document independent of whether the DTD is known or not.

The first line of our sample document,

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

is the so-called *XML declaration*. It expresses that the document follows the conventions of XML version 1.0, and that the document is encoded using characters from the ISO-8859-1 character set (often known as "Latin 1", mostly used in Western Europe). Although the XML declaration is not mandatory, it is good style to include it; everybody sees at the first glance that the document uses XML markup and not the similar-looking HTML and SGML markup languages. If you omit the XML declaration, the parser will assume that the document is encoded as UTF-8 or UTF-16 (there is a rule that makes it possible to distinguish between UTF-8 and UTF-16 automatically); these are encodings of Unicode's universal character set. (Note that PXP, unlike its predecessor "Markup", fully supports Unicode.)

The second line,

```
<!DOCTYPE document SYSTEM "simple.dtd">
```

names the DTD that is going to be used for the rest of the document. In general, it is possible that the DTD consists of two parts, the so-called external and the internal subset. "External" means that the DTD exists as a second file; "internal" means that the DTD is included in the same file. In this example, there is only an external subset, and the system identifier "simple.dtd" specifies where the DTD file can be found. System identifiers are interpreted as URLs; for instance this would be legal:

```
<!DOCTYPE document SYSTEM "http://host/location/simple.dtd">
```

Please note that PXP cannot interpret HTTP identifiers by default, but it is possible to change the interpretation of system identifiers.

The word immediately following `DOCTYPE` determines which of the declared element types (here "document", "section", and "paragraph") is used for the outermost element, the *root element*. In this example it is `document` because the outermost element is delimited by `<document>` and `</document>`.

The DTD consists of three declarations for element types: `document`, `section`, and `paragraph`. Such a declaration has two parts:

```
<!ELEMENT name content-model>
```

The content model is a regular expression which describes the possible inner structure of the element. Here, `document` contains one or more sections, and a `section` contains one or more paragraphs. Note that these two element types are not allowed to contain arbitrary text. Only the `paragraph` element type is declared such that parsed character data (indicated by the symbol `#PCDATA`) is permitted.

See below for a detailed discussion of content models.

1.1.2. XML parsers and processors

XML documents are human-readable, but this is not the main purpose of this language. XML has been designed such that documents can be read by a program called an *XML parser*. The parser checks that the document is well-formatted, and it represents the document as objects of the programming language. There are two aspects when checking the document: First, the document must follow some basic syntactic rules, such as that tags are written in angle brackets, that for every start tag there must be a corresponding end tag and so on. A document respecting these rules is *well-formed*. Second, the document must match the DTD in which case the document is *valid*. Many parsers check only on well-formedness and ignore the DTD; PXP is designed such that it can even validate the document.

A parser does not make a sensible application, it only reads XML documents. The whole application working with XML-formatted data is called an *XML processor*. Often XML processors convert documents into another format, such as HTML or Postscript. Sometimes processors extract data of the documents and output the processed data again XML-formatted. The parser can help the application processing the document; for example it can provide means to access the document in a specific manner. PXP supports an object-oriented access layer specially.

1.1.3. Discussion

As we have seen, there are two levels of description: On the one hand, XML can define rules about the format of a document (the DTD), on the other hand, XML expresses structured documents. There are a number of possible applications:

- XML can be used to express structured texts. Unlike HTML, there is no canonical interpretation; one would have to write a backend for the DTD that translates the structured texts into a format that existing browsers, printers etc. understand. The advantage of a self-defined document format is that it is possible to design the format in a more problem-oriented way. For example, if the task is to extract reports from a database, one can use a DTD that reflects the structure of the report or the database. A possible approach would be to have an element type for every database table and for every column. Once the DTD has been designed, the report procedure can be splitted up in a part that selects the database rows and outputs them as an XML document according to the DTD, and in a part that translates the document into other formats. Of course, the latter part can be solved in a generic way, e.g. there may be configurable backends for all DTDs that follow the approach and have element types for tables and columns.

  XML plays the role of a configurable intermediate format. The database extraction function can be written without having to know the details of typesetting; the backends can be written without having to know the details of the database.

  Of course, there are traditional solutions. One can define an ad hoc intermediate text file format. This disadvantage is that there are no names for the pieces of the format, and that such formats usually lack of documentation because of this. Another solution would be to have a binary representation, either as language-dependent or language-independent structure (example of the latter can be found in RPC implementations). The disadvantage is that it is harder to view such representations, one has to write pretty printers for this purpose. It is also more difficult to enter test data; XML is plain text that can be written using an arbitrary editor (Emacs has even a good XML mode, PSGML). All these alternatives suffer from a missing structure checker, i.e. the programs processing these formats usually do not check the input file or input object in detail; XML parsers check the syntax of the input (the so-called well-formedness check), and the advanced parsers like PXP even verify that the structure matches the DTD (the so-called validation).

- XML can be used as configurable communication language. A fundamental problem of every communication is that sender and receiver must follow the same conventions about the language. For data exchange, the question is usually which data records and fields are available, how they are syntactically composed, and which values are possible for the various fields. Similar questions arise for text document exchange. XML does not answer these problems completely, but it reduces the number of ambiguities for such conventions: The outlines of the syntax are specified by the DTD (but not necessarily the details), and XML introduces canonical names for the components of documents such that it is simpler to describe the rest of the syntax and the semantics informally.

- XML is a data storage format. Currently, every software product tends to use its own way to store data; commercial software often does not describe such formats, and it is a pain to integrate such software into a bigger project. XML can help to improve this situation when several applications share the same syntax of data files. DTDs are then neutral instances that check the format of data files independent of applications.

1.2. Highlights of XML

This section explains many of the features of XML, but not all, and some features not in detail. For a complete description, see the XML specification (http://www.w3.org/TR/1998/REC-xml-19980210.html).

1.2.1. The DTD and the instance

The DTD contains various declarations; in general you can only use a feature if you have previously declared it. The document instance file may contain the full DTD, but it is also possible to split the DTD into an internal and an external subset. A document must begin as follows if the full DTD is included:

```
<?xml version="1.0" encoding="Your encoding"?>
<!DOCTYPE root [
  Declarations
]>
```

These declarations are called the *internal subset*. Note that the usage of entities and conditional sections is restricted within the internal subset.

If the declarations are located in a different file, you can refer to this file as follows:

```
<?xml version="1.0" encoding="Your encoding"?>
<!DOCTYPE root SYSTEM "file name">
```

The declarations in the file are called the *external subset*. The file name is called the *system identifier*. It is also possible to refer to the file by a so-called *public identifier*, but most XML applications won't use this feature.

You can also specify both internal and external subsets. In this case, the declarations of both subsets are mixed, and if there are conflicts, the declaration of the internal subset overrides those of the external subset with the same name. This looks as follows:

```
<?xml version="1.0" encoding="Your encoding"?>
<!DOCTYPE root  SYSTEM "file name" [
```

```
  Declarations
]>
```

The XML declaration (the string beginning with `<?xml` and ending at `?>`) should specify the encoding of the file. Common values are UTF-8, and the ISO-8859 series of character sets. Note that every file parsed by the XML processor can begin with an XML declaration and that every file may have its own encoding.

The name of the root element must be mentioned directly after the `DOCTYPE` string. This means that a full document instance looks like

```
<?xml version="1.0" encoding="Your encoding"?>
<!DOCTYPE root  SYSTEM "file name" [
  Declarations
]>

<root>
  inner contents
</root>
```

1.2.2. Reserved characters

Some characters are generally reserved to indicate markup such that they cannot be used for character data. These characters are <, >, and &. Furthermore, single and double quotes are sometimes reserved. If you want to include such a character as character, write it as follows:

- `<` instead of <
- `>` instead of >
- `&` instead of &
- `'` instead of '
- `"` instead of "

All other characters are free in the document instance. It is possible to include a character by its position in the Unicode alphabet:

```
&#n;
```

where *n* is the decimal number of the character. Alternatively, you can specify the character by its hexadecimal number:

```
&#xn;
```

In the scope of declarations, the character % is no longer free. To include it as character, you must use the notations `%` or `%`.

Note that besides <, >, &, ', and " there are no predefines character entities. This is different from HTML which defines a list of characters that can be referenced by name (e.g. ä for ä); however, if you prefer named characters, you can declare such entities yourself (see below).

1.2.3. Elements and ELEMENT declarations

Elements structure the document instance in a hierarchical way. There is a top-level element, the *root element*, which contains a sequence of inner elements and character sections. The inner elements are structured in the same way. Every element has an *element type*. The beginning of the element is indicated by a *start tag*, written

```
<element-type>
```

and the element continues until the corresponding *end tag* is reached:

```
</element-type>
```

In XML, it is not allowed to omit start or end tags, even if the DTD would permit this. Note that there are no special rules how to interpret spaces or newlines near start or end tags; all spaces and newlines count.

Every element type must be declared before it can be used. The declaration consists of two parts: the ELEMENT declaration describes the content model, i.e. which inner elements are allowed; the ATTLIST declaration describes the attributes of the element.

An element can simply allow everything as content. This is written:

```
<!ELEMENT name ANY>
```

On the opposite, an element can be forced to be empty; declared by:

```
<!ELEMENT name EMPTY>
```

Note that there is an abbreviated notation for empty element instances: `<name/>`.

There are two more sophisticated forms of declarations: so-called *mixed declarations*, and *regular expressions*. An element with mixed content contains character data interspersed with inner elements, and the set of allowed inner elements can be specified. In contrast to this, a regular expression declaration does not allow character data, but the inner elements can be described by the more powerful means of regular expressions.

A declaration for mixed content looks as follows:

<!ELEMENT *name* (#PCDATA | *element*$_1$ | ... | *element*$_n$)*>

or if you do not want to allow any inner element, simply

<!ELEMENT *name* (#PCDATA)>

Example

If element type `q` is declared as

```
<!ELEMENT q (#PCDATA | r | s)*>
```

this is a legal instance:

```
<q>This is character data<r></r>with <s></s>inner elements</q>
```

But this is illegal because `t` has not been enumerated in the declaration:

```
<q>This is character data<r></r>with <t></t>inner elements</q>
```

The other form uses a regular expression to describe the possible contents:

<!ELEMENT *name regexp*>

The following well-known regexp operators are allowed:

- *element-name*
- (*subexpr*$_1$, ..., *subexpr*$_n$)
- (*subexpr*$_1$ | ... | *subexpr*$_n$)
- *subexpr**
- *subexpr*+
- *subexpr*?

The `,` operator indicates a sequence of sub-models, the | operator describes alternative sub-models. The `*` indicates zero or more repetitions, and `+` one or more repetitions. Finally, `?` can be used for optional sub-models. As atoms the regexp can contain names of elements; note that it is not allowed to include `#PCDATA`.

The exact syntax of the regular expressions is rather strange. This can be explained best by a list of constraints:

- The outermost expression must not be *element-name*.

Illegal: `<!ELEMENT x y>`; this must be written as `<!ELEMENT x (y)>`.

- For the unary operators *`subexpr`*`*`, *`subexpr`*`+`, and *`subexpr`*`?`, the *`subexpr`* must not be again an unary operator.

  Illegal: `<!ELEMENT x y**>`; this must be written as `<!ELEMENT x (y*)*>`.

- Between `)` and one of the unary operatory `*`, `+`, or `?`, there must not be whitespace.

  Illegal: `<!ELEMENT x (y|z) *>`; this must be written as `<!ELEMENT x (y|z)*>`.

- There is the additional constraint that the right parenthsis must be contained in the same entity as the left parenthesis; see the section about parsed entities below.

Note that there is another restriction on regular expressions which must be deterministic. This means that the parser must be able to see by looking at the next token which alternative is actually used, or whether the repetition stops. The reason for this is simply compatability with SGML (there is no intrinsic reason for this rule; XML can live without this restriction).

Example

The elements are declared as follows:

```
<!ELEMENT q (r?, (s | t)+)>
<!ELEMENT r (#PCDATA)>
<!ELEMENT s EMPTY>
<!ELEMENT t (q | r)>
```

This is a legal instance:

```
<q><r>Some characters</r><s/></q>
```

(Note: `<s/>` is an abbreviation for `<s></s>`.) It would be illegal to leave `<s/>` out because at least one instance of `s` or `t` must be present. It would be illegal, too, if characters existed outside the `r` element; the only exception is white space. – This is legal, too:

```
<q><s/><t><q><s/></q></t></q>
```

1.2.4. Attribute lists and ATTLIST declarations

Elements may have attributes. These are put into the start tag of an element as follows:

<*element-name attribute*$_1$="*value*$_1$" ... *attribute*$_n$="*value*$_n$">

Instead of "*value*$_k$" it is also possible to use single quotes as in '*value*$_k$'. Note that you cannot use double quotes literally within the value of the attribute if double quotes are the delimiters; the same applies to single quotes. You can generally not use < and & as characters in attribute values. It is possible to include the paraphrases <, >, &, ', and " (and any other reference to a general entity as long as the entity is not defined by an external file) as well as &#*n*;.

Before you can use an attribute you must declare it. An ATTLIST declaration looks as follows:

```
<!ATTLIST element-name
          attribute-name attribute-type attribute-default
          ...
          attribute-name attribute-type attribute-default
>
```

There are a lot of types, but most important are:

- `CDATA`: Every string is allowed as attribute value.
- `NMTOKEN`: Every nametoken is allowed as attribute value. Nametokens consist (mainly) of letters, digits, ., :, -, _ in arbitrary order.
- `NMTOKENS`: A space-separated list of nametokens is allowed as attribute value.

The most interesting default declarations are:

- `#REQUIRED`: The attribute must be specified.
- `#IMPLIED`: The attribute can be specified but also can be left out. The application can find out whether the attribute was present or not.
- `"value"` or `'value'`: This particular value is used as default if the attribute is omitted in the element.

Example

This is a valid attribute declaration for element type `r`:

```
<!ATTLIST r
          x CDATA    #REQUIRED
          y NMTOKEN  #IMPLIED
          z NMTOKENS "one two three">
```

This means that `x` is a required attribute that cannot be left out, while `y` and `z` are optional. The XML parser indicates the application whether `y` is present or not, but if `z` is missing the default value "one two three" is returned automatically.

This is a valid example of these attributes:

```
<r x="He said: &quot;I don't like quotes!&quot;" y='1'>
```

1.2.5. Parsed entities

Elements describe the logical structure of the document, while *entities* determine the physical structure. Entities are the pieces of text the parser operates on, mostly files and macros. Entities may be *parsed* in which case the parser reads the text and interprets it as XML markup, or *unparsed* which simply means that the data of the entity has a foreign format (e.g. a GIF icon).

If the parsed entity is going to be used as part of the DTD, it is called a *parameter entity*. You can declare a parameter entity with a fixed text as content by:

```
<!ENTITY % name "value">
```

Within the DTD, you can *refer to* this entity, i.e. read the text of the entity, by:

```
%name;
```

Such entities behave like macros, i.e. when they are referred to, the macro text is inserted and read instead of the original text.

Example

For example, you can declare two elements with the same content model by:

```
<!ENTITY % model "a | b | c">
<!ELEMENT x (%model;)>
<!ELEMENT y (%model;)>
```

If the contents of the entity are given as string constant, the entity is called an *internal* entity. It is also possible to name a file to be used as content (an *external* entity):

```
<!ENTITY % name SYSTEM "file name">
```

There are some restrictions for parameter entities:

- If the internal parameter entity contains the first token of a declaration (i.e. `<!`), it must also contain the last token of the declaration, i.e. the `>`. This means that the entity either contains a whole number of complete declarations, or some text from the middle of one declaration.

  Illegal:

  ```
  <!ENTITY % e "(a | b | c)>">
  <!ELEMENT x %e;
  ```

  Because `<!` is contained in the main entity, and the corresponding `>` is contained in the entity `e`.

- If the internal parameter entity contains a left paranthesis, it must also contain the corresponding right paranthesis.

  Illegal:

  ```
  <!ENTITY % e "(a | b | c">
  <!ELEMENT x %e;)>
  ```

  Because `(` is contained in the entity `e`, and the corresponding `)` is contained in the main entity.

- When reading text from an entity, the parser automatically inserts one space character before the entity text and one space character after the entity text. However, this rule is not applied within the definition of another entity.

  Legal:

  ```
  <!ENTITY % suffix "gif">
  <!ENTITY iconfile 'icon.%suffix;'>
  ```

  Because `%suffix;` is referenced within the definition text for `iconfile`, no additional spaces are added.

  Illegal:

  ```
  <!ENTITY % suffix "test">
  <!ELEMENT x.%suffix; ANY>
  ```

  Because `%suffix;` is referenced outside the definition text of another entity, the parser replaces `%suffix;` by *space*`test`*space*.

  Illegal:

  ```
  <!ENTITY % e "(a | b | c)">
  <!ELEMENT x %e;*>
  ```

  Because there is a whitespace between `)` and `*`, which is illegal.

- An external parameter entity must always consist of a whole number of complete declarations.
- In the internal subset of the DTD, a reference to a parameter entity (internal or external) is only allowed at positions where a new declaration can start.

If the parsed entity is going to be used in the document instance, it is called a *general entity*. Such entities can be used as abbreviations for frequent phrases, or to include external files. Internal general entities are declared as follows:

```
<!ENTITY name "value">
```

External general entities are declared this way:

```
<!ENTITY name SYSTEM "file name">
```

References to general entities are written as:

&*name*;

The main difference between parameter and general entities is that the former are only recognized in the DTD and that the latter are only recognized in the document instance. As the DTD is parsed before the document, the parameter entities are expanded first; for example it is possible to use the content of a parameter entity as the name of a general entity: &%name;;[1].

General entities must respect the element hierarchy. This means that there must be an end tag for every start tag in the entity value, and that end tags without corresponding start tags are not allowed.

Example

If the author of a document changes sometimes, it is worthwhile to set up a general entity containing the names of the authors. If the author changes, you need only to change the definition of the entity, and do not need to check all occurrences of authors' names:

```
<!ENTITY authors "Gerd Stolpmann">
```

In the document text, you can now refer to the author names by writing &authors;.

Illegal: The following two entities are illegal because the elements in the definition do not nest properly:

```
<!ENTITY lengthy-tag "<section textcolor='white' background='graphic'>">
<!ENTITY nonsense    "<a></b>">
```

Earlier in this introduction we explained that there are substitutes for reserved characters: <, >, &, ', and ". These are simply predefined general entities; note that they are the only predefined entities. It is allowed to define these entities again as long as the meaning is unchanged.

1.2.6. Notations and unparsed entities

Unparsed entities have a foreign format and can thus not be read by the XML parser. Unparsed entities are always external. The format of an unparsed entity must have been declared, such a format is called a *notation*. The entity can then be declared by referring to this notation. As unparsed entities do not contain XML text, it is not possible to include them directly into the document; you can only declare attributes such that names of unparsed entities are acceptable values.

As you can see, unparsed entities are too complicated in order to have any purpose. It is almost always better to simply pass the name of the data file as normal attribute value, and let the application recognize and process the foreign format.

1.3. A complete example: The *readme* DTD

The reason for *readme* was that I often wrote two versions of files such as README and INSTALL which explain aspects of a distributed software archive; one version was ASCII-formatted, the other was written in HTML. Maintaining both versions means double amount of work, and changes of one version may be forgotten in the other version. To improve this situation I invented the *readme* DTD which allows me to maintain only one source written as XML document, and to generate the ASCII and the HTML version from it.

In this section, I explain only the DTD. The *readme* DTD is contained in the PXP distribution together with the two converters to produce ASCII and HTML. Another section of this manual describes the HTML converter.

The documents have a simple structure: There are up to three levels of nested sections, paragraphs, item lists, footnotes, hyperlinks, and text emphasis. The outermost element has usually the type `readme`, it is declared by

```
<!ELEMENT readme (sect1+)>
<!ATTLIST readme
          title CDATA #REQUIRED>
```

This means that this element contains one or more sections of the first level (element type `sect1`), and that the element has a required attribute `title` containing character data (CDATA). Note that `readme` elements must not contain text data.

The three levels of sections are declared as follows:

```
<!ELEMENT sect1 (title,(sect2|p|ul)+)>

<!ELEMENT sect2 (title,(sect3|p|ul)+)>

<!ELEMENT sect3 (title,(p|ul)+)>
```

Every section has a `title` element as first subelement. After the title an arbitrary but non-empty sequence of inner sections, paragraphs and item lists follows. Note that the inner sections must belong to the next higher section level; `sect3` elements must not contain inner sections because there is no next higher level.

Obviously, all three declarations allow paragraphs (`p`) and item lists (`ul`). The definition can be simplified at this point by using a parameter entity:

```
<!ENTITY % p.like "p|ul">

<!ELEMENT sect1 (title,(sect2|%p.like;)+)>
```

```
<!ELEMENT sect2 (title,(sect3|%p.like;)+)>
```

```
<!ELEMENT sect3 (title,(%p.like;)+)>
```

Here, the entity `p.like` is nothing but a macro abbreviating the same sequence of declarations; if new elements on the same level as `p` and `ul` are later added, it is sufficient only to change the entity definition. Note that there are some restrictions on the usage of entities in this context; most important, entities containing a left paranthesis must also contain the corresponding right paranthesis.

Note that the entity `p.like` is a *parameter* entity, i.e. the ENTITY declaration contains a percent sign, and the entity is referred to by `%p.like;`. This kind of entity must be used to abbreviate parts of the DTD; the *general* entities declared without percent sign and referred to as `&name;` are not allowed in this context.

The `title` element specifies the title of the section in which it occurs. The title is given as character data, optionally interspersed with line breaks (`br`):

```
<!ELEMENT title (#PCDATA|br)*>
```

Compared with the `title` *attribute* of the `readme` element, this element allows inner markup (i.e. `br`) while attribute values do not: It is an error if an attribute value contains the left angle bracket < literally such that it is impossible to include inner elements.

The paragraph element `p` has a structure similar to `title`, but it allows more inner elements:

```
<!ENTITY % text "br|code|em|footnote|a">
```

```
<!ELEMENT p (#PCDATA|%text;)*>
```

Line breaks do not have inner structure, so they are declared as being empty:

```
<!ELEMENT br EMPTY>
```

This means that really nothing is allowed within `br`; you must always write `<br></br>` or abbreviated `<br/>`.

Code samples should be marked up by the `code` tag; emphasized text can be indicated by `em`:

```
<!ELEMENT code (#PCDATA)>
```

```
<!ELEMENT em (#PCDATA|%text;)*>
```

That `code` elements are not allowed to contain further markup while `em` elements do is a design decision by the author of the DTD.

Unordered lists simply consists of one or more list items, and a list item may contain paragraph-level material:

```
<!ELEMENT ul (li+)>

<!ELEMENT li (%p.like;)*>
```

Footnotes are described by the text of the note; this text may contain text-level markup. There is no mechanism to describe the numbering scheme of footnotes, or to specify how footnote references are printed.

```
<!ELEMENT footnote (#PCDATA|%text;)*>
```

Hyperlinks are written as in HTML. The anchor tag contains the text describing where the link points to, and the `href` attribute is the pointer (as URL). There is no way to describe locations of "hash marks". If the link refers to another *readme* document, the attribute `readmeref` should be used instead of `href`. The reason is that the converted document has usually a different system identifier (file name), and the link to a converted document must be converted, too.

```
<!ELEMENT a (#PCDATA)*>
<!ATTLIST a
          href      CDATA #IMPLIED
          readmeref CDATA #IMPLIED
>
```

Note that although it is only sensible to specify one of the two attributes, the DTD has no means to express this restriction.

So far the DTD. Finally, here is a document for it:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE readme SYSTEM "readme.dtd">
<readme title="How to use the readme converters">
<sect1>
  <title>Usage</title>
  <p>
    The <em>readme</em> converter is invoked on the command line by:
  </p>
  <p>
    <code>readme [ -text | -html ] input.xml</code>
  </p>
  <p>
    Here a list of options:
  </p>
  <ul>
    <li>
      <p><code>-
text</code>: specifies that ASCII output should be produced</p>
```

```
    </li>
    <li>
      <p><code>-
html</code>: specifies that HTML output should be produced</p>
    </li>
  </ul>
  <p>
    The input file must be given on the command line. The converted output is
    printed to <em>stdout</em>.
  </p>
</sect1>
<sect1>
  <title>Author</title>
  <p>
    The program has been written by
    <a href="mailto:Gerd.Stolpmann@darmstadt.netsurf.de">Gerd Stolpmann</a>.
  </p>
</sect1>
</readme>
```

Notes

1. This construct is only allowed within the definition of another entity; otherwise extra spaces would be added (as explained above). Such indirection is not recommended.

   Complete example:

   ```
   <!ENTITY % variant "a">      <!- or "b" ->
   <!ENTITY text-a "This is text A.">
   <!ENTITY text-b "This is text B.">
   <!ENTITY text "&#38;text-%variant;;">
   ```

   You can now write `&text;` in the document instance, and depending on the value of `variant` either `text-a` or `text-b` is inserted.

Chapter 2. Using PXP

2.1. Validation

The parser can be used to *validate* a document. This means that all the constraints that must hold for a valid document are actually checked. Validation is the default mode of PXP, i.e. every document is validated while it is being parsed.

In the `examples` directory of the distribution you find the `pxpvalidate` application. It is invoked in the following way:

```
pxpvalidate [ -wf ] file...
```

The files mentioned on the command line are validated, and every warning and every error messages are printed to stderr.

The -wf switch modifies the behaviour such that a well-formedness parser is simulated. In this mode, the ELEMENT, ATTLIST, and NOTATION declarations of the DTD are ignored, and only the ENTITY declarations will take effect. This mode is intended for documents lacking a DTD. Please note that the parser still scans the DTD fully and will report all errors in the DTD; such checks are not required by a well-formedness parser.

The `pxpvalidate` application is the simplest sensible program using PXP, you may consider it as "hello world" program.

2.2. How to parse a document from an application

Let me first give a rough overview of the object model of the parser. The following items are represented by objects:

- *Documents:* The document representation is more or less the anchor for the application; all accesses to the parsed entities start here. It is described by the class `document` contained in the module `Pxp_document`. You can get some global information, such as the XML declaration the document begins with, the DTD of the document, global processing instructions, and most important, the document tree.
- *The contents of documents:* The contents have the structure of a tree: Elements contain other elements and text[1]. The common type to represent both kinds of content is `node` which is a class type that unifies the properties of elements and character data. Every node has a list of children (which is empty

if the element is empty or the node represents text); nodes may have attributes; nodes have always text contents. There are two implementations of `node`, the class `element_impl` for elements, and the class `data_impl` for text data. You find these classes and class types in the module `Pxp_document`, too.

Note that attribute lists are represented by non-class values.

- *The node extension:* For advanced usage, every node of the document may have an associated *extension* which is simply a second object. This object must have the three methods `clone`, `node`, and `set_node` as bare minimum, but you are free to add methods as you want. This is the preferred way to add functionality to the document tree[2]. The class type `extension` is defined in `Pxp_document`, too.
- *The DTD:* Sometimes it is necessary to access the DTD of a document; the average application does not need this feature. The class `dtd` describes DTDs, and makes it possible to get representations of element, entity, and notation declarations as well as processing instructions contained in the DTD. This class, and `dtd_element`, `dtd_notation`, and `proc_instruction` can be found in the module `Pxp_dtd`. There are a couple of classes representing different kinds of entities; these can be found in the module `Pxp_entity`.

Additionally, the following modules play a role:

- *Pxp_yacc:* Here the main parsing functions such as `parse_document_entity` are located. Some additional types and functions allow the parser to be configured in a non-standard way.
- *Pxp_types:* This is a collection of basic types and exceptions.

There are some further modules that are needed internally but are not part of the API.

Let the document to be parsed be stored in a file called `doc.xml`. The parsing process is started by calling the function

```
val parse_document_entity : config -> source -> 'ext spec -> 'ext document
```

defined in the module `Pxp_yacc`. The first argument specifies some global properties of the parser; it is recommended to start with the `default_config`. The second argument determines where the document to be parsed comes from; this may be a file, a channel, or an entity ID. To parse `doc.xml`, it is sufficient to pass `from_file "doc.xml"`.

The third argument passes the object specification to use. Roughly speaking, it determines which classes implement the node objects of which element types, and which extensions are to be used. The `'ext` polymorphic variable is the type of the extension. For the moment, let us simply pass `default_spec` as this argument, and ignore it.

So the following expression parses `doc.xml`:

```
open Pxp_yacc
let d = parse_document_entity default_config (from_file "doc.xml") de-
fault_spec
```

Note that default_config implies that warnings are collected but not printed. Errors raise one of the exception defined in Pxp_types; to get readable errors and warnings catch the exceptions as follows:

```
class warner =
  object
    method warn w =
      print_endline ("WARNING: " ^ w)
  end
;;

try
  let config = { default_config with warner = new warner } in
  let d = parse_document_entity config (from_file "doc.xml") default_spec
  in
    ...
with
   e ->
     print_endline (Pxp_types.string_of_exn e)
```

Now d is an object of the document class. If you want the node tree, you can get the root element by

```
let root = d # root
```

and if you would rather like to access the DTD, determine it by

```
let dtd = d # dtd
```

As it is more interesting, let us investigate the node tree now. Given the root element, it is possible to recursively traverse the whole tree. The children of a node n are returned by the method sub_nodes, and the type of a node is returned by node_type. This function traverses the tree, and prints the type of each node:

```
let rec print_structure n =
  let ntype = n # node_type in
  match ntype with
    T_element name ->
      print_endline ("Element of type " ^ name);
      let children = n # sub_nodes in
      List.iter print_structure children
  | T_data ->
      print_endline "Data"
  | _ ->
      (* Other node types are not possible unless the parser is configured
         differently.
       *)
```

```
    assert false
```

You can call this function by

```
print_structure root
```

The type returned by `node_type` is either `T_element name` or `T_data`. The `name` of the element type is the string included in the angle brackets. Note that only elements have children; data nodes are always leaves of the tree.

There are some more methods in order to access a parsed node tree:

- `n # parent`: Returns the parent node, or raises `Not_found` if the node is already the root
- `n # root`: Returns the root of the node tree.
- `n # attribute a`: Returns the value of the attribute with name `a`. The method returns a value for every *declared* attribute, independently of whether the attribute instance is defined or not. If the attribute is not declared, `Not_found` will be raised. (In well-formedness mode, every attribute is considered as being implicitly declared with type `CDATA`.)

  The following return values are possible: `Value s`, `Valuelist sl` , and `Implied_value`. The first two value types indicate that the attribute value is available, either because there is a definition `a="value"` in the XML text, or because there is a default value (declared in the DTD). Only if both the instance definition and the default declaration are missing, the latter value `Implied_value` will be returned.

  In the DTD, every attribute is typed. There are single-value types (CDATA, ID, IDREF, ENTITY, NMTOKEN, enumerations), in which case the method passes `Value s` back, where `s` is the normalized string value of the attribute. The other types (IDREFS, ENTITIES, NMTOKENS) represent list values, and the parser splits the XML literal into several tokens and returns these tokens as `Valuelist sl`.

  Normalization means that entity references (the `&name;` tokens) and character references (`&#number;`) are replaced by the text they represent, and that white space characters are converted into plain spaces.

- `n # data`: Returns the character data contained in the node. For data nodes, the meaning is obvious as this is the main content of data nodes. For element nodes, this method returns the concatenated contents of all inner data nodes.

  Note that entity references included in the text are resolved while they are being parsed; for example the text "a <> b" will be returned as "a <> b" by this method. Spaces of data nodes are always preserved. Newlines are preserved, but always converted to \n characters even if newlines are encoded as \r\n or \r. Normally you will never see two adjacent data nodes because the parser collapses all data material at one location into one node. (However, if you create your own tree or transform the parsed tree, it is possible to have adjacent data nodes.)

Note that elements that do *not* allow #PCDATA as content will not have data nodes as children. This means that spaces and newlines, the only character material allowed for such elements, are silently dropped.

For example, if the task is to print all contents of elements with type "valuable" whose attribute "priority" is "1", this function can help:

```
let rec print_valuable_prio1 n =
  let ntype = n # node_type in
  match ntype with
    T_element "valuable" when n # attribute "priority" = Value "1" ->
      print_endline "Valuable node with priotity 1 found:";
      print_endline (n # data)
  | (T_element _ | T_data) ->
      let children = n # sub_nodes in
      List.iter print_valuable_prio1 children
  | _ ->
      assert false
```

You can call this function by:

```
print_valuable_prio1 root
```

If you like a DSSSL-like style, you can make the function `process_children` explicit:

```
let rec print_valuable_prio1 n =

  let process_children n =
    let children = n # sub_nodes in
    List.iter print_valuable_prio1 children
  in

  let ntype = n # node_type in
  match ntype with
    T_element "valuable" when n # attribute "priority" = Value "1" ->
      print_endline "Valuable node with priority 1 found:";
      print_endline (n # data)
  | (T_element _ | T_data) ->
      process_children n
  | _ ->
      assert false
```

So far, O'Caml is now a simple "style-sheet language": You can form a big "match" expression to distinguish between all significant cases, and provide different reactions on different conditions. But this technique has limitations; the "match" expression tends to get larger and larger, and it is difficult to store intermediate values as there is only one big recursion. Alternatively, it is also possible to represent the various cases as classes, and to use dynamic method lookup to find the appropiate class. The next section explains this technique in detail.

2.3. Class-based processing of the node tree

By default, the parsed node tree consists of objects of the same class; this is a good design as long as you want only to access selected parts of the document. For complex transformations, it may be better to use different classes for objects describing different element types.

For example, if the DTD declares the element types `a`, `b`, and `c`, and if the task is to convert an arbitrary document into a printable format, the idea is to define for every element type a separate class that has a method `print`. The classes are `eltype_a`, `eltype_b`, and `eltype_c`, and every class implements `print` such that elements of the type corresponding to the class are converted to the output format.

The parser supports such a design directly. As it is impossible to derive recursive classes in O'Caml[3], the specialized element classes cannot be formed by simply inheriting from the built-in classes of the parser and adding methods for customized functionality. To get around this limitation, every node of the document tree is represented by *two* objects, one called "the node" and containing the recursive definition of the tree, one called "the extension". Every node object has a reference to the extension, and the extension has a reference to the node. The advantage of this model is that it is now possible to customize the extension without affecting the typing constraints of the recursive node definition.

Every extension must have the three methods `clone`, `node`, and `set_node`. The method `clone` creates a deep copy of the extension object and returns it; `node` returns the node object for this extension object; and `set_node` is used to tell the extension object which node is associated with it, this method is automatically called when the node tree is initialized. The following definition is a good starting point for these methods; usually `clone` must be further refined when instance variables are added to the class:

```
class custom_extension =
  object (self)

    val mutable node = (None : custom_extension node option)

    method clone = {< >}
    method node =
      match node with
          None ->
            assert false
        | Some n -> n
    method set_node n =
      node <- Some n
```

```
  end
```

This part of the extension is usually the same for all classes, so it is a good idea to consider `custom_extension` as the super-class of the further class definitions. Continuining the example of above, we can define the element type classes as follows:

```
class virtual custom_extension =
  object (self)
    ... clone, node, set_node defined as above ...

    method virtual print : out_channel -> unit
  end

class eltype_a =
  object (self)
    inherit custom_extension
    method print ch = ...
  end

class eltype_b =
  object (self)
    inherit custom_extension
    method print ch = ...
  end

class eltype_c =
  object (self)
    inherit custom_extension
    method print ch = ...
  end
```

The method `print` can now be implemented for every element type separately. Note that you get the associated node by invoking

```
self # node
```

and you get the extension object of a node `n` by writing

```
n # extension
```

It is guaranteed that

```
self # node # extension == self
```

always holds.

Here are sample definitions of the `print` methods:

```
class eltype_a =
  object (self)
    inherit custom_extension
    method print ch =
      (* Nodes <a>...</a> are only containers: *)
      output_string ch "(";
      List.iter
        (fun n -> n # extension # print ch)
        (self # node # sub_nodes);
      output_string ch ")";
  end

class eltype_b =
  object (self)
    inherit custom_extension
    method print ch =
      (* Print the value of the CDATA attribute "print": *)
      match self # node # attribute "print" with
        Value s       -> output_string ch s
      | Implied_value -> output_string ch "<missing>"
      | Valuelist l   -> assert false
                         (* not possible because the att is CDATA *)
  end

class eltype_c =
  object (self)
    inherit custom_extension
    method print ch =
      (* Print the contents of this element: *)
      output_string ch (self # node # data)
  end

class null_extension =
  object (self)
    inherit custom_extension
    method print ch = assert false
  end
```

The remaining task is to configure the parser such that these extension classes are actually used. Here another problem arises: It is not possible to dynamically select the class of an object to be created. As workaround, PXP allows the user to specify *exemplar objects* for the various element types; instead of creating the nodes of the tree by applying the `new` operator the nodes are produced by duplicating the exemplars. As object duplication preserves the class of the object, one can create fresh objects of every class for which previously an exemplar has been registered.

Exemplars are meant as objects without contents, the only interesting thing is that exemplars are instances of a certain class. The creation of an exemplar for an element node can be done by:

```
let element_exemplar = new element_impl extension_exemplar
```

And a data node exemplar is created by:

```
let data_exemplar = new data_impl extension_exemplar
```

The classes `element_impl` and `data_impl` are defined in the module `Pxp_document`. The constructors initialize the fresh objects as empty objects, i.e. without children, without data contents, and so on. The `extension_exemplar` is the initial extension object the exemplars are associated with.

Once the exemplars are created and stored somewhere (e.g. in a hash table), you can take an exemplar and create a concrete instance (with contents) by duplicating it. As user of the parser you are normally not concerned with this as this is part of the internal logic of the parser, but as background knowledge it is worthwhile to mention that the two methods `create_element` and `create_data` actually perform the duplication of the exemplar for which they are invoked, additionally apply modifications to the clone, and finally return the new object. Moreover, the extension object is copied, too, and the new node object is associated with the fresh extension object. Note that this is the reason why every extension object must have a `clone` method.

The configuration of the set of exemplars is passed to the `parse_document_entity` function as third argument. In our example, this argument can be set up as follows:

```
let spec =
  make_spec_from_alist
    ~data_exemplar:            (new data_impl (new null_extension))
    ~default_element_exemplar: (new element_impl (new null_extension))
    ~element_alist:
       [ "a",  new element_impl (new eltype_a);
         "b",  new element_impl (new eltype_b);
         "c",  new element_impl (new eltype_c);
       ]
    ()
```

The `~element_alist` function argument defines the mapping from element types to exemplars as associative list. The argument `~data_exemplar` specifies the exemplar for data nodes, and the `~default_element_exemplar` is used whenever the parser finds an element type for which the associative list does not define an exemplar.

The configuration is now complete. You can still use the same parsing functions, only the initialization is a bit different. For example, call the parser by:

```
let d = parse_document_entity default_config (from_file "doc.xml") spec
```

Note that the resulting document `d` has a usable type; especially the `print` method we added is visible. So you can print your document by

```
d # root # extension # print stdout
```

This object-oriented approach looks rather complicated; this is mostly caused by working around some problems of the strict typing system of O'Caml. Some auxiliary concepts such as extensions were needed, but the practical consequences are low. In the next section, one of the examples of the distribution is explained, a converter from *readme* documents to HTML.

2.4. Example: An HTML backend for the *readme* DTD

The converter from *readme* documents to HTML documents follows strictly the approach to define one class per element type. The HTML code is similar to the *readme* source, because of this most elements can be converted in the following way: Given the input element

```
<e>content</e>
```

the conversion text is the concatenation of a computed prefix, the recursively converted content, and a computed suffix.

Only one element type cannot be handled by this scheme: `footnote`. Footnotes are collected while they are found in the input text, and they are printed after the main text has been converted and printed.

2.4.1. Header

```
open Pxp_types
open Pxp_document
```

2.4.2. Type declarations

```
class type footnote_printer =
```

```
  object
    method footnote_to_html : store_type -> out_channel -> unit
  end

and store_type =
  object
    method alloc_footnote : footnote_printer -> int
    method print_footnotes : out_channel -> unit
  end
;;
```

2.4.3. Class store

The store is a container for footnotes. You can add a footnote by invoking alloc_footnote; the argument is an object of the class footnote_printer, the method returns the number of the footnote. The interesting property of a footnote is that it can be converted to HTML, so a footnote_printer is an object with a method footnote_to_html. The class footnote which is defined below has a compatible method footnote_to_html such that objects created from it can be used as footnote_printers.

The other method, print_footnotes prints the footnotes as definition list, and is typically invoked after the main material of the page has already been printed. Every item of the list is printed by footnote_to_html.

```
class store =
  object (self)

    val mutable footnotes = ( [] : (int * footnote_printer) list )
    val mutable next_footnote_number = 1

    method alloc_footnote n =
      let number = next_footnote_number in
      next_footnote_number <- number+1;
      footnotes <- footnotes @ [ number, n ];
      number

    method print_footnotes ch =
      if footnotes <> [] then begin
output_string ch "<hr align=left noshade=noshade width=\"30%\">\n";
output_string ch "<dl>\n";
List.iter
  (fun (_,n) ->
```

```
     n # footnote_to_html (self : #store_type :> store_type) ch)
  footnotes;
output_string ch "</dl>\n";
      end

  end
;;
```

2.4.4. Function escape_html

This function converts the characters <, >, &, and " to their HTML representation. For example, escape_html "<>" = "<>". Other characters are left unchanged.

```
let escape_html s =
  Str.global_substitute
    (Str.regexp "<\\|>\\|&\\|\"")
    (fun s ->
      match Str.matched_string s with
        "<" -> "&lt;"
      | ">" -> "&gt;"
      | "&" -> "&amp;"
      | "\"" -> "&quot;"
      | _ -> assert false)
    s
;;
```

2.4.5. Virtual class shared

This virtual class is the abstract superclass of the extension classes shown below. It defines the standard methods clone, node, and set_node, and declares the type of the virtual method to_html. This method recursively traverses the whole element tree, and prints the converted HTML code to the output channel passed as second argument. The first argument is the reference to the global store object which collects the footnotes.

```
class virtual shared =
  object (self)

    (* -- default_ext -- *)

    val mutable node = (None : shared node option)
```

```
    method clone = {< >}
    method node =
      match node with
          None ->
            assert false
        | Some n -> n
    method set_node n =
      node <- Some n

    (* -- virtual -- *)

    method virtual to_html : store -> out_channel -> unit

  end
;;
```

2.4.6. Class `only_data`

This class defines `to_html` such that the character data of the current node is converted to HTML. Note that `self` is an extension object, `self # node` is the node object, and `self # node # data` returns the character data of the node.

```
class only_data =
  object (self)
    inherit shared

    method to_html store ch =
      output_string ch (escape_html (self # node # data))
  end
;;
```

2.4.7. Class `readme`

This class converts elements of type `readme` to HTML. Such an element is (by definition) always the root element of the document. First, the HTML header is printed; the `title` attribute of the element determines the title of the HTML page. Some aspects of the HTML page can be configured by setting certain parameter entities, for example the background color, the text color, and link colors. After the header, the `body` tag, and the headline have been printed, the contents of the page are converted by invoking `to_html` on all children of the current node (which is the root node). Then, the footnotes are appended to this by telling the global `store` object to print the footnotes. Finally, the end tags of the HTML pages are printed.

This class is an example how to access the value of an attribute: The value is determined by invoking `self # node # attribute "title"`. As this attribute has been declared as CDATA and as being required, the value has always the form `Value s` where `s` is the string value of the attribute.

You can also see how entity contents can be accessed. A parameter entity object can be looked up by `self # node # dtd # par_entity "name"`, and by invoking `replacement_text` the value of the entity is returned after inner parameter and character entities have been processed. Note that you must use `gen_entity` instead of `par_entity` to access general entities.

```
class readme =
  object (self)
    inherit shared

    method to_html store ch =
      (* output header *)
      output_string
ch "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 3.2 Final//EN\">";
      output_string
ch "<!- WARNING! This is a generated file, do not edit! ->\n";
      let title =
match self # node # attribute "title" with
    Value s -> s
  | _ -> assert false
      in
      let html_header, _ =
try (self # node # dtd # par_entity "readme:html:header")
            # replacement_text
with WF_error _ -> "", false in
      let html_trailer, _ =
try (self # node # dtd # par_entity "readme:html:trailer")
            # replacement_text
with WF_error _ -> "", false in
      let html_bgcolor, _ =
try (self # node # dtd # par_entity "readme:html:bgcolor")
            # replacement_text
with WF_error _ -> "white", false in
      let html_textcolor, _ =
try (self # node # dtd # par_entity "readme:html:textcolor")
            # replacement_text
with WF_error _ -> "", false in
      let html_alinkcolor, _ =
```

```
try (self # node # dtd # par_entity "readme:html:alinkcolor")
            # replacement_text
with WF_error _ -> "", false in
      let html_vlinkcolor, _ =
try (self # node # dtd # par_entity "readme:html:vlinkcolor")
            # replacement_text
with WF_error _ -> "", false in
      let html_linkcolor, _ =
try (self # node # dtd # par_entity "readme:html:linkcolor")
            # replacement_text
with WF_error _ -> "", false in
      let html_background, _ =
try (self # node # dtd # par_entity "readme:html:background")
            # replacement_text
with WF_error _ -> "", false in

      output_string ch "<html><header><title>\n";
      output_string ch (escape_html title);
      output_string ch "</title></header>\n";
      output_string ch "<body ";
      List.iter
(fun (name,value) ->
   if value <> "" then
     output_string ch (name ^ "=\"" ^ escape_html value ^ "\" "))
[ "bgcolor",    html_bgcolor;
  "text",       html_textcolor;
  "link",       html_linkcolor;
  "alink",      html_alinkcolor;
  "vlink",      html_vlinkcolor;
];
      output_string ch ">\n";
      output_string ch html_header;
      output_string ch "<h1>";
      output_string ch (escape_html title);
      output_string ch "</h1>\n";
      (* process main content: *)
      List.iter
(fun n -> n # extension # to_html store ch)
(self # node # sub_nodes);
      (* now process footnotes *)
      store # print_footnotes ch;
      (* trailer *)
      output_string ch html_trailer;
      output_string ch "</html>\n";
```

```
  end
;;
```

2.4.8. Classes `section`, `sect1`, `sect2`, and `sect3`

As the conversion process is very similar, the conversion classes of the three section levels are derived from the more general `section` class. The HTML code of the section levels only differs in the type of the headline, and because of this the classes describing the section levels can be computed by replacing the class argument `the_tag` of `section` by the HTML name of the headline tag.

Section elements are converted to HTML by printing a headline and then converting the contents of the element recursively. More precisely, the first sub-element is always a `title` element, and the other elements are the contents of the section. This structure is declared in the DTD, and it is guaranteed that the document matches the DTD. Because of this the title node can be separated from the rest without any checks.

Both the title node, and the body nodes are then converted to HTML by calling `to_html` on them.

```
class section the_tag =
  object (self)
    inherit shared

    val tag = the_tag

    method to_html store ch =
      let sub_nodes = self # node # sub_nodes in
      match sub_nodes with
  title_node :: rest ->
    output_string ch ("<" ^ tag ^ ">\n");
    title_node # extension # to_html store ch;
    output_string ch ("\n</" ^ tag ^ ">");
    List.iter
      (fun n -> n # extension # to_html store ch)
      rest
| _ ->
    assert false
  end
;;

class sect1 = section "h1";;
class sect2 = section "h3";;
class sect3 = section "h4";;
```

2.4.9. Classes map_tag, p, em, ul, li

Several element types are converted to HTML by simply mapping them to corresponding HTML element types. The class map_tag implements this, and the class argument the_target_tag determines the tag name to map to. The output consists of the start tag, the recursively converted inner elements, and the end tag.

```
class map_tag the_target_tag =
  object (self)
    inherit shared

    val target_tag = the_target_tag

    method to_html store ch =
      output_string ch ("<" ^ target_tag ^ ">\n");
      List.iter
(fun n -> n # extension # to_html store ch)
(self # node # sub_nodes);
      output_string ch ("\n</" ^ target_tag ^ ">");
  end
;;

class p = map_tag "p";;
class em = map_tag "b";;
class ul = map_tag "ul";;
class li = map_tag "li";;
```

2.4.10. Class br

Element of type br are mapped to the same HTML type. Note that HTML forbids the end tag of br.

```
class br =
  object (self)
    inherit shared

    method to_html store ch =
      output_string ch "<br>\n";
      List.iter
(fun n -> n # extension # to_html store ch)
(self # node # sub_nodes);
  end
;;
```

2.4.11. Class code

The code type is converted to a pre section (preformatted text). As the meaning of tabs is unspecified in HTML, tabs are expanded to spaces.

```
class code =
  object (self)
    inherit shared

    method to_html store ch =
      let data = self # node # data in
      (* convert tabs *)
      let l = String.length data in
      let rec preprocess i column =
(* this is very ineffective but comprehensive: *)
if i < l then
  match data.[i] with
      '\t' ->
let n = 8 - (column mod 8) in
String.make n ' ' ^ preprocess (i+1) (column + n)
    | '\n' ->
"\n" ^ preprocess (i+1) 0
    | c ->
String.make 1 c ^ preprocess (i+1) (column + 1)
else
  ""
      in
      output_string ch "<p><pre>";
      output_string ch (escape_html (preprocess 0 0));
      output_string ch "</pre></p>";

  end
;;
```

2.4.12. Class a

Hyperlinks, expressed by the a element type, are converted to the HTML a type. If the target of the hyperlink is given by href, the URL of this attribute can be used directly. Alternatively, the target can be given by readmeref in which case the ".html" suffix must be added to the file name.

Note that within a only #PCDATA is allowed, so the contents can be converted directly by applying escape_html to the character data contents.

```
class a =
  object (self)
    inherit shared

    method to_html store ch =
      output_string ch "<a ";
      let href =
match self # node # attribute "href" with
    Value v -> escape_html v
  | Valuelist _ -> assert false
  | Implied_value ->
      begin match self # node # attribute "readmeref" with
  Value v -> escape_html v ^ ".html"
| Valuelist _ -> assert false
| Implied_value ->
    ""
      end
      in
      if href <> "" then
output_string ch ("href=\""  ^ href ^ "\"");
      output_string ch ">";
      output_string ch (escape_html (self # node # data));
      output_string ch "</a>";

  end
;;
```

2.4.13. Class `footnote`

The `footnote` class has two methods: `to_html` to convert the footnote reference to HTML, and `footnote_to_html` to convert the footnote text itself.

The footnote reference is converted to a local hyperlink; more precisely, to two anchor tags which are connected with each other. The text anchor points to the footnote anchor, and the footnote anchor points to the text anchor.

The footnote must be allocated in the `store` object. By allocating the footnote, you get the number of the footnote, and the text of the footnote is stored until the end of the HTML page is reached when the footnotes can be printed. The `to_html` method stores simply the object itself, such that the `footnote_to_html` method is invoked on the same object that encountered the footnote.

The `to_html` only allocates the footnote, and prints the reference anchor, but it does not print nor convert the contents of the note. This is deferred until the footnotes actually get printed, i.e. the recursive call of `to_html` on the sub nodes is done by `footnote_to_html`.

Note that this technique does not work if you make another footnote within a footnote; the second footnote gets allocated but not printed.

```
class footnote =
  object (self)
    inherit shared

    val mutable footnote_number = 0

    method to_html store ch =
      let number =
store # alloc_footnote (self : #shared :> footnote_printer) in
      let foot_anchor =
"footnote" ^ string_of_int number in
      let text_anchor =
"textnote" ^ string_of_int number in
      footnote_number <- number;
      output_string ch ( "<a name=\"" ^ text_anchor ^ "\" href=\"#" ^
 foot_anchor ^ "\">[" ^ string_of_int number ^
 "]</a>" )

    method footnote_to_html store ch =
      (* prerequisite: we are in a definition list <dl>...</dl> *)
      let foot_anchor =
"footnote" ^ string_of_int footnote_number in
      let text_anchor =
"textnote" ^ string_of_int footnote_number in
      output_string ch ("<dt><a name=\"" ^ foot_anchor ^ "\" href=\"#" ^
text_anchor ^ "\">[" ^ string_of_int footnote_number ^
"]</a></dt>\n<dd>");
      List.iter
(fun n -> n # extension # to_html store ch)
(self # node # sub_nodes);
      output_string ch ("\n</dd>")

  end
;;
```

2.4.14. The specification of the document model

This code sets up the hash table that connects element types with the exemplars of the extension classes that convert the elements to HTML.

```
open Pxp_yacc

let tag_map =
  make_spec_from_alist
    ~data_exemplar:(new data_impl (new only_data))
    ~default_element_exemplar:(new element_impl (new no_markup))
    ~element_alist:
      [ "readme", (new element_impl (new readme));
"sect1",  (new element_impl (new sect1));
"sect2",  (new element_impl (new sect2));
"sect3",  (new element_impl (new sect3));
"title",  (new element_impl (new no_markup));
"p",      (new element_impl (new p));
"br",     (new element_impl (new br));
"code",   (new element_impl (new code));
"em",     (new element_impl (new em));
"ul",     (new element_impl (new ul));
"li",     (new element_impl (new li));
"footnote", (new element_impl (new footnote : #shared :> shared));
"a",      (new element_impl (new a));
      ]
    ()
;;
```

Notes

1. Elements may also contain processing instructions. Unlike other document models, PXP separates processing instructions from the rest of the text and provides a second interface to access them (method `pinstr`). However, there is a parser option (`enable_pinstr_nodes`) which changes the behaviour of the parser such that extra nodes for processing instructions are included into the tree.

   Furthermore, the tree does normally not contain nodes for XML comments; they are ignored by default. Again, there is an option (`enable_comment_nodes`) changing this.

2. Due to the typing system it is more or less impossible to derive recursive classes in O'Caml. To get around this, it is common practice to put the modifiable or extensible part of recursive objects into parallel objects.

3. The problem is that the subclass is usually not a subtype in this case because O'Caml has a contravariant subtyping rule.

Chapter 3. The objects representing the document

This description might be out-of-date. See the module interface files for updated information.

3.1. The document class

```
class [ 'ext ] document :
  Pxp_types.collect_warnings ->
  object
    method init_xml_version : string -> unit
    method init_root : 'ext node -> unit

    method xml_version : string
    method xml_standalone : bool
    method dtd : dtd
    method root : 'ext node

    method encoding : Pxp_types.rep_encoding

    method add_pinstr : proc_instruction -> unit
    method pinstr : string -> proc_instruction list
    method pinstr_names : string list

    method write : Pxp_types.output_stream -> Pxp_types.encoding -> unit

  end
;;
```

The methods beginning with `init_` are only for internal use of the parser.

- `xml_version`: returns the version string at the beginning of the document. For example, "1.0" is returned if the document begins with `<?xml version="1.0"?>`.
- `xml_standalone`: returns the boolean value of `standalone` declaration in the XML declaration. If the `standalone` attribute is missing, `false` is returned.
- `dtd`: returns a reference to the global DTD object.
- `root`: returns a reference to the root element.
- `encoding`: returns the internal encoding of the document. This means that all strings of which the document consists are encoded in this character set.

- `pinstr`: returns the processing instructions outside the DTD and outside the root element. The argument passed to the method names a *target*, and the method returns all instructions with this target. The target is the first word inside `<?` and `?>`.
- `pinstr_names`: returns the names of the processing instructions
- `add_pinstr`: adds another processing instruction. This method is used by the parser itself to enter the instructions returned by `pinstr`, but you can also enter additional instructions.
- `write`: writes the document to the passed stream as XML text using the passed (external) encoding. The generated text is always valid XML and can be parsed by PXP; however, the text is badly formatted (this is not a pretty printer).

3.2. The class type `node`

From `Pxp_document`:

```
type node_type =
  T_data
| T_element of string
| T_super_root
| T_pinstr of string
| T_comment
and some other, reserved types
;;

class type [ 'ext ] node =
  object ('self)
    constraint 'ext = 'ext node #extension

    (* General observers *)

    method extension : 'ext
    method dtd : dtd
    method parent : 'ext node
    method root : 'ext node
    method sub_nodes : 'ext node list
    method iter_nodes : ('ext node -> unit) -> unit
    method iter_nodes_sibl :
           ('ext node option -> 'ext node -> 'ext node option -> unit) -
> unit
    method node_type : node_type
    method encoding : Pxp_types.rep_encoding
    method data : string
    method position : (string * int * int)
```

```
method comment : string option
method pinstr : string -> proc_instruction list
method pinstr_names : string list
method write : Pxp_types.output_stream -> Pxp_types.encoding -> unit

(* Attribute observers *)

method attribute : string -> Pxp_types.att_value
method required_string_attribute : string -> string
method optional_string_attribute : string -> string option
method required_list_attribute : string -> string list
method optional_list_attribute : string -> string list
method attribute_names : string list
method attribute_type : string -> Pxp_types.att_type
method attributes : (string * Pxp_types.att_value) list
method id_attribute_name : string
method id_attribute_value : string
method idref_attribute_names : string

(* Modifying methods *)

method add_node : ?force:bool -> 'ext node -> unit
method add_pinstr : proc_instruction -> unit
method delete : unit
method set_nodes : 'ext node list -> unit
method quick_set_attributes : (string * Pxp_types.att_value) list -> unit
method set_comment : string option -> unit

(* Cloning methods *)

method orphaned_clone : 'self
method orphaned_flat_clone : 'self
method create_element :
          ?position:(string * int * int) ->
          dtd -> node_type -> (string * string) list ->
              'ext node
method create_data : dtd -> string -> 'ext node
method keep_always_whitespace_mode : unit

(* Validating methods *)

method local_validate : ?use_dfa:bool -> unit -> unit

(* ... Internal methods are undocumented. *)
```

```
  end
;;
```

In the module `Pxp_types` you can find another type definition that is important in this context:

```
type Pxp_types.att_value =
    Value     of string
  | Valuelist of string list
  | Implied_value
;;
```

3.2.1. The structure of document trees

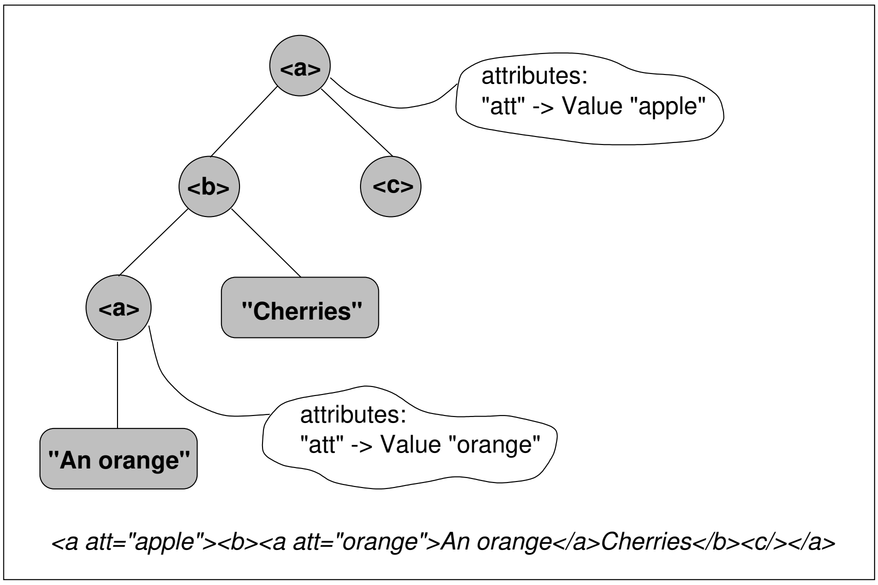
A node represents either an element or a character data section. There are two classes implementing the two aspects of nodes: `element_impl` and `data_impl`. The latter class does not implement all methods because some methods do not make sense for data nodes.

(Note: PXP also supports a mode which forces that processing instructions and comments are represented as nodes of the document tree. However, these nodes are instances of `element_impl` with node types `T_pinstr` and `T_comment`, respectively. This mode must be explicitly configured; the basic representation knows only element and data nodes.)

The following figure (*A tree with element nodes, data nodes, and attributes*) shows an example how a tree is constructed from element and data nodes. The circular areas represent element nodes whereas the ovals denote data nodes. Only elements may have subnodes; data nodes are always leaves of the tree. The subnodes of an element can be either element or data nodes; in both cases the O'Caml objects storing the nodes have the class type `node`.

Attributes (the clouds in the picture) are not directly integrated into the tree; there is always an extra link to the attribute list. This is also true for processing instructions (not shown in the picture). This means that there are separated access methods for attributes and processing instructions.

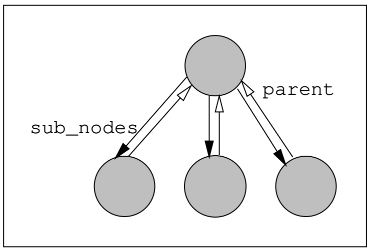
Figure 3-1. A tree with element nodes, data nodes, and attributes



Only elements, data sections, attributes and processing instructions (and comments, if configured) can, directly or indirectly, occur in the document tree. It is impossible to add entity references to the tree; if the parser finds such a reference, not the reference as such but the referenced text (i.e. the tree representing the structured text) is included in the tree.

Note that the parser collapses as much data material into one data node as possible such that there are normally never two adjacent data nodes. This invariant is enforced even if data material is included by entity references or CDATA sections, or if a data sequence is interrupted by comments. So `a & b <- comment -> c <![CDATA[ <> d]]>` is represented by only one data node, for instance. However, you can create document trees manually which break this invariant; it is only the way the parser forms the tree.

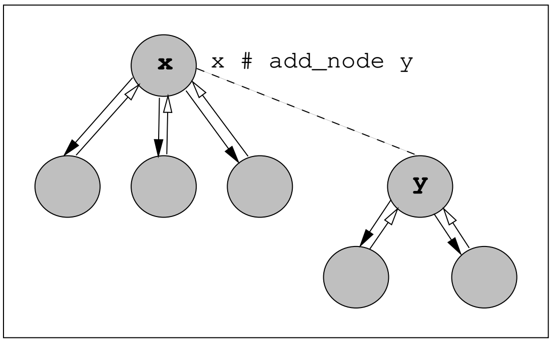
Figure 3-2. Nodes are doubly linked trees

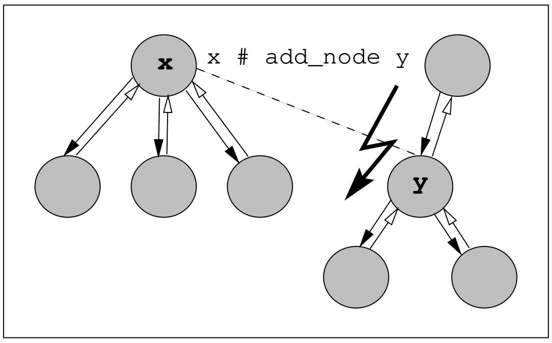


The node tree has links in both directions: Every node has a link to its parent (if any), and it has links to the subnodes (see figure *Nodes are doubly linked trees*). Obviously, this doubly-linked structure simplifies the navigation in the tree; but has also some consequences for the possible operations on trees.

Because every node must have at most *one* parent node, operations are illegal if they violate this condition. The following figure (*A node can only be added if it is a root*) shows on the left side that node `y` is added to `x` as new subnode which is allowed because `y` does not have a parent yet. The right side of the picture illustrates what would happen if `y` had a parent node; this is illegal because `y` would have two parents after the operation.

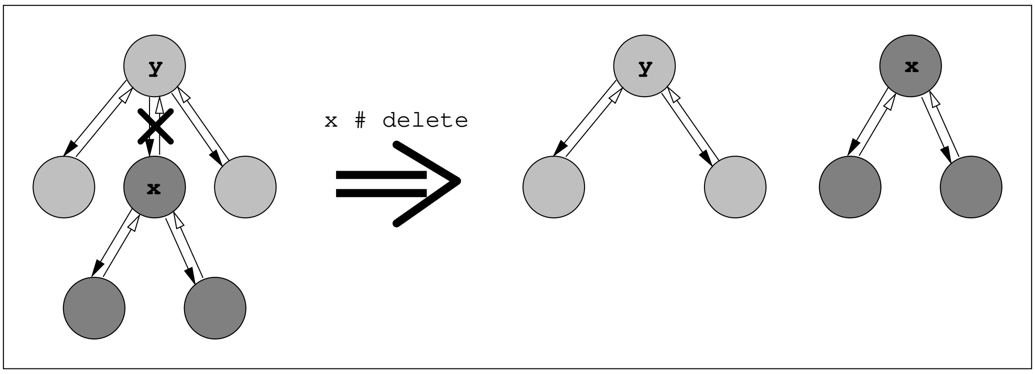
Figure 3-3. A node can only be added if it is a root





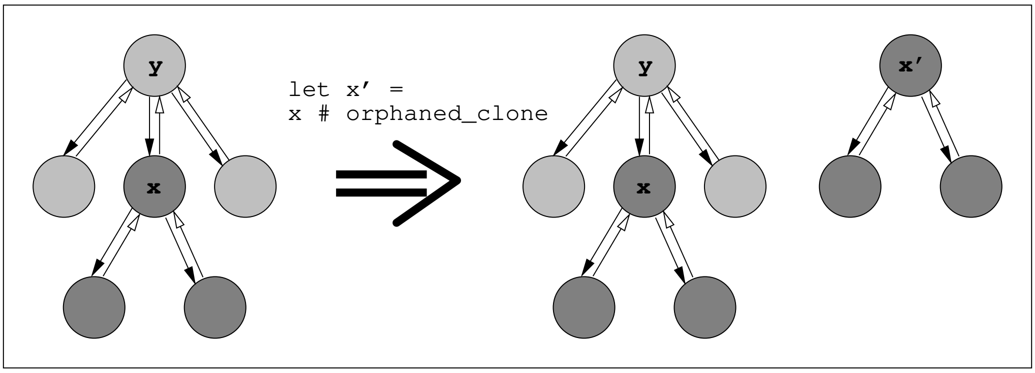
The "delete" operation simply removes the links between two nodes. In the picture (*A deleted node becomes the root of the subtree*) the node `x` is deleted from the list of subnodes of `y`. After that, `x` becomes the root of the subtree starting at this node.

Figure 3-4. A deleted node becomes the root of the subtree



It is also possible to make a clone of a subtree; illustrated in *The clone of a subtree*. In this case, the clone is a copy of the original subtree except that it is no longer a subnode. Because cloning never keeps the connection to the parent, the clones are called *orphaned*.

Figure 3-5. The clone of a subtree



3.2.2. The methods of the class type `node`

General observers .

- `extension`: The reference to the extension object which belongs to this node (see ...).
- `dtd`: Returns a reference to the global DTD. All nodes of a tree must share the same DTD.
- `parent`: Get the father node. Raises `Not_found` in the case the node does not have a parent, i.e. the node is the root.
- `root`: Gets the reference to the root node of the tree. Every node is contained in a tree with a root, so this method always succeeds. Note that this method *searches* the root, which costs time proportional to the length of the path to the root.
- `sub_nodes`: Returns references to the children. The returned list reflects the order of the children. For data nodes, this method returns the empty list.
- `iter_nodes f`: Iterates over the children, and calls `f` for every child in turn.
- `iter_nodes_sibl f`: Iterates over the children, and calls `f` for every child in turn. `f` gets as arguments the previous node, the current node, and the next node.
- `node_type`: Returns either `T_data` which means that the node is a data node, or `T_element n` which means that the node is an element of type `n`. If configured, possible node types are also `T_pinstr t` indicating that the node represents a processing instruction with target `t`, and `T_comment` in which case the node is a comment.
- `encoding`: Returns the encoding of the strings.
- `data`: Returns the character data of this node and all children, concatenated as one string. The encoding of the string is what the method `encoding` returns. - For data nodes, this method simply returns the represented characters. For elements, the meaning of the method has been extended such that it returns something useful, i.e. the effectively contained characters, without markup. (For `T_pinstr` and `T_comment` nodes, the method returns the empty string.)
- `position`: If configured, this method returns the position of the element as triple (entity, line, byteposition). For data nodes, the position is not stored. If the position is not available the triple `"?", 0, 0` is returned.
- `comment`: Returns `Some text` for comment nodes, and `None` for other nodes. The `text` is everything between the comment delimiters <– and –>.
- `pinstr n`: Returns all processing instructions that are directly contained in this element and that have a *target* specification of `n`. The target is the first word after the `<?`.
- `pinstr_names`: Returns the list of all targets of processing instructions directly contained in this element.
- `write s enc`: Prints the node and all subnodes to the passed output stream as valid XML text, using the passed external encoding.

Attribute observers .

- `attribute n`: Returns the value of the attribute with name `n`. This method returns a value for every declared attribute, and it raises `Not_found` for any undeclared attribute. Note that it even returns a value if the attribute is actually missing but is declared as `#IMPLIED` or has a default value. - Possible values are:

- `Implied_value`: The attribute has been declared with the keyword `#IMPLIED`, and the attribute is missing in the attribute list of this element.
- `Value s`: The attribute has been declared as type `CDATA`, as `ID`, as `IDREF`, as `ENTITY`, or as `NMTOKEN`, or as enumeration or notation, and one of the two conditions holds: (1) The attribute value is present in the attribute list in which case the value is returned in the string `s`. (2) The attribute has been omitted, and the DTD declared the attribute with a default value. The default value is returned in `s`. - Summarized, `Value s` is returned for non-implied, non-list attribute values.
- `Valuelist l`: The attribute has been declared as type `IDREFS`, as `ENTITIES`, or as `NMTOKENS`, and one of the two conditions holds: (1) The attribute value is present in the attribute list in which case the space-separated tokens of the value are returned in the string list `l`. (2) The attribute has been omitted, and the DTD declared the attribute with a default value. The default value is returned in `l`. - Summarized, `Valuelist l` is returned for all list-type attribute values.

Note that before the attribute value is returned, the value is normalized. This means that newlines are converted to spaces, and that references to character entities (i.e. `&#`*`n`*`;`) and general entities (i.e. `&`*`name`*`;`) are expanded; if necessary, expansion is performed recursively.

In well-formedness mode, there is no DTD which could declare an attribute. Because of this, every occuring attribute is considered as a CDATA attribute.

- `required_string_attribute n`: returns the Value attribute called n, or the Valuelist attribute as a string where the list elements are separated by spaces. If the attribute value is implied, or if the attribute does not exists, the method will fail. - This method is convenient if you expect a non-implied and non-list attribute value.
- `optional_string_attribute n`: returns the Value attribute called n, or the Valuelist attribute as a string where the list elements are separated by spaces. If the attribute value is implied, or if the attribute does not exists, the method returns None. - This method is convenient if you expect a non-list attribute value including the implied value.
- `required_list_attribute n`: returns the Valuelist attribute called n, or the Value attribute as a list with a single element. If the attribute value is implied, or if the attribute does not exists, the method will fail. - This method is convenient if you expect a list attribute value.
- `optional_list_attribute n`: returns the Valuelist attribute called n, or the Value attribute as a list with a single element. If the attribute value is implied, or if the attribute does not exists, an empty list will be returned. - This method is convenient if you expect a list attribute value or the implied value.
- `attribute_names`: returns the list of all attribute names of this element. As this is a validating parser, this list is equal to the list of declared attributes.
- `attribute_type n`: returns the type of the attribute called `n`. See the module `Pxp_types` for a description of the encoding of the types.
- `attributes`: returns the list of pairs of names and values for all attributes of this element.

- `id_attribute_name`: returns the name of the attribute that is declared with type ID. There is at most one such attribute. The method raises `Not_found` if there is no declared ID attribute for the element type.
- `id_attribute_value`: returns the value of the attribute that is declared with type ID. There is at most one such attribute. The method raises `Not_found` if there is no declared ID attribute for the element type.
- `idref_attribute_names`: returns the list of attribute names that are declared as IDREF or IDREFS.

Modifying methods . The following methods are only defined for element nodes (more exactly: the methods are defined for data nodes, too, but fail always).

- `add_node sn`: Adds sub node `sn` to the list of children. This operation is illustrated in the picture *A node can only be added if it is a root*. This method expects that `sn` is a root, and it requires that `sn` and the current object share the same DTD.

  Because `add_node` is the method the parser itself uses to add new nodes to the tree, it performs by default some simple validation checks: If the content model is a regular expression, it is not allowed to add data nodes to this node unless the new nodes consist only of whitespace. In this case, the new data nodes are silently dropped (you can change this by invoking `keep_always_whitespace_mode`).

  If the document is flagged as stand-alone, these data nodes only containing whitespace are even forbidden if the element declaration is contained in an external entity. This case is detected and rejected.

  If the content model is `EMPTY`, it is not allowed to add any data node unless the data node is empty. In this case, the new data node is silently dropped.

  These checks only apply if there is a DTD. In well-formedness mode, it is assumed that every element is declared with content model `ANY` which prohibits any validation check. Furthermore, you turn these checks off by passing `~force:true` as first argument.

- `add_pinstr pi`: Adds the processing instruction `pi` to the list of processing instructions.
- `delete`: Deletes this node from the tree. After this operation, this node is no longer the child of the former father node; and the node loses the connection to the father as well. This operation is illustrated by the figure *A deleted node becomes the root of the subtree*.
- `set_nodes nl`: Sets the list of children to `nl`. It is required that every member of `nl` is a root, and that all members and the current object share the same DTD. Unlike `add_node`, no validation checks are performed.
- `quick_set_attributes atts`: sets the attributes of this element to `atts`. It is *not* checked whether `atts` matches the DTD or not; it is up to the caller of this method to ensure this. (This method may be useful to transform the attribute values, i.e. apply a mapping to every attribute.)
- `set_comment text`: This method is only applicable to `T_comment` nodes; it sets the comment text contained by such nodes.

Cloning methods .

- `orphaned_clone`: Returns a clone of the node and the complete tree below this node (deep clone). The clone does not have a parent (i.e. the reference to the parent node is *not* cloned). While copying the subtree, strings are skipped; it is likely that the original tree and the copy tree share strings. Extension objects are cloned by invoking the `clone` method on the original objects; how much of the extension objects is cloned depends on the implemention of this method.

  This operation is illustrated by the figure *The clone of a subtree*.
- `orphaned_flat_clone`: Returns a clone of the node, but sets the list of sub nodes to [], i.e. the sub nodes are not cloned.
- `create_element dtd nt al`: Returns a flat copy of this node (which must be an element) with the following modifications: The DTD is set to `dtd`; the node type is set to `nt`, and the new attribute list is set to `al` (given as list of (name,value) pairs). The copy does not have children nor a parent. It does not contain processing instructions. See the example below.

  Note that you can specify the position of the new node by the optional argument `~position`.
- `create_data dtd cdata`: Returns a flat copy of this node (which must be a data node) with the following modifications: The DTD is set to `dtd`; the node type is set to `T_data`; the attribute list is empty (data nodes never have attributes); the list of children and PIs is empty, too (same reason). The new node does not have a parent. The value `cdata` is the new character content of the node. See the example below.
- `keep_always_whitespace_mode`: Even data nodes which are normally dropped because they only contain ignorable whitespace, can added to this node once this mode is turned on. (This mode is useful to produce canonical XML.)

Validating methods . There is one method which locally validates the node, i.e. checks whether the subnodes match the content model of this node.

- `local_validate`: Checks that this node conforms to the DTD by comparing the type of the subnodes with the content model for this node. (Applications need not call this method unless they add new nodes themselves to the tree.)

3.2.3. The class `element_impl`

This class is an implementation of `node` which realizes element nodes:

```
class [ 'ext ] element_impl : 'ext -> [ 'ext ] node
```

Constructor. You can create a new instance by

```
new element_impl extension_object
```

which creates a special form of empty element which already contains a reference to the *extension_object*, but is otherwise empty. This special form is called an *exemplar*. The purpose of exemplars is that they serve as patterns that can be duplicated and filled with data. The method `create_element` is designed to perform this action.

Example. First, create an exemplar by

```
let exemplar_ext = ... in
let exemplar     = new element_impl exemplar_ext in
```

The `exemplar` is not used in node trees, but only as a pattern when the element nodes are created:

```
let element = exemplar # create_element dtd (T_element name) attlist
```

The `element` is a copy of `exemplar` (even the extension `exemplar_ext` has been copied) which ensures that `element` and its extension are objects of the same class as the exemplars; note that you need not to pass a class name or other meta information. The copy is initially connected with the `dtd`, it gets a node type, and the attribute list is filled. The `element` is now fully functional; it can be added to another element as child, and it can contain references to subnodes.

3.2.4. The class `data_impl`

This class is an implementation of `node` which should be used for all character data nodes:

```
class [ 'ext ] data_impl : 'ext -> [ 'ext ] node
```

Constructor. You can create a new instance by

```
new data_impl extension_object
```

which creates an empty exemplar node which is connected to *extension_object*. The node does not contain a reference to any DTD, and because of this it cannot be added to node trees.

To get a fully working data node, apply the method `create_data` to the exemplar (see example).

Example. First, create an exemplar by

```
let exemplar_ext = ... in
let exemplar     = new exemplar_ext data_impl in
```

The `exemplar` is not used in node trees, but only as a pattern when the data nodes are created:

```
let data_node = exemplar # create_data dtd "The characters con-
tained in the data node"
```

The `data_node` is a copy of `exemplar`. The copy is initially connected with the `dtd`, and it is filled with character material. The `data_node` is now fully functional; it can be added to an element as child.

3.2.5. The type `spec`

The type `spec` defines a way to handle the details of creating nodes from exemplars.

```
type 'ext spec
constraint 'ext = 'ext node #extension

val make_spec_from_mapping :
      ?super_root_exemplar : 'ext node ->
      ?comment_exemplar : 'ext node ->
      ?default_pinstr_exemplar : 'ext node ->
      ?pinstr_mapping : (string, 'ext node) Hashtbl.t ->
      data_exemplar: 'ext node ->
      default_element_exemplar: 'ext node ->
      element_mapping: (string, 'ext node) Hashtbl.t ->
      unit ->
        'ext spec

val make_spec_from_alist :
      ?super_root_exemplar : 'ext node ->
      ?comment_exemplar : 'ext node ->
      ?default_pinstr_exemplar : 'ext node ->
      ?pinstr_alist : (string * 'ext node) list ->
      data_exemplar: 'ext node ->
      default_element_exemplar: 'ext node ->
      element_alist: (string * 'ext node) list ->
      unit ->
        'ext spec
```

The two functions `make_spec_from_mapping` and `make_spec_from_alist` create `spec` values. Both functions are functionally equivalent and the only difference is that the first function prefers hashtables and the latter associative lists to describe mappings from names to exemplars.

You can specify exemplars for the various kinds of nodes that need to be generated when an XML document is parsed:

- `~super_root_exemplar`: This exemplar is used to create the super root. This special node is only created if the corresponding configuration option has been selected; it is the parent node of the root node which may be convenient if every working node must have a parent.
- `~comment_exemplar`: This exemplar is used when a comment node must be created. Note that such nodes are only created if the corresponding configuration option is "on".
- `~default_pinstr_exemplar`: If a node for a processing instruction must be created, and the instruction is not listed in the table passed by `~pinstr_mapping` or `~pinstr_alist`, this exemplar is used. Again the configuration option must be "on" in order to create such nodes at all.
- `~pinstr_mapping` or `~pinstr_alist`: Map the target names of processing instructions to exemplars. These mappings are only used when nodes for processing instructions are created.
- `~data_exemplar`: The exemplar for ordinary data nodes.
- `~default_element_exemplar`: This exemplar is used if an element node must be created, but the element type cannot be found in the tables `element_mapping` or `element_alist`.
- `~element_mapping` or `~element_alist`: Map the element types to exemplars. These mappings are used to create element nodes.

In most cases, you only want to create `spec` values to pass them to the parser functions found in `Pxp_yacc`. However, it might be useful to apply `spec` values directly.

The following functions create various types of nodes by selecting the corresponding exemplar from the passed `spec` value, and by calling `create_element` or `create_data` on the exemplar.

```
val create_data_node :
      'ext spec ->
      dtd ->
      (* data material: *) string ->
          'ext node

val create_element_node :
      ?position:(string * int * int) ->
      'ext spec ->
      dtd ->
      (* element type: *) string ->
      (* attributes: *) (string * string) list ->
          'ext node

val create_super_root_node :
      ?position:(string * int * int) ->
      'ext spec ->
       dtd ->
           'ext node

val create_comment_node :
      ?position:(string * int * int) ->
```

```
        'ext spec ->
        dtd ->
        (* comment text: *) string ->
            'ext node

val create_pinstr_node :
        ?position:(string * int * int) ->
        'ext spec ->
        dtd ->
        proc_instruction ->
            'ext node
```

3.2.6. Examples

Building trees. Here is the piece of code that creates the tree of the figure *A tree with element nodes, data nodes, and attributes*. The extension object and the DTD are beyond the scope of this example.

```
let exemplar_ext = ... (* some extension *) in
let dtd = ... (* some DTD *) in

let element_exemplar = new element_impl exemplar_ext in
let data_exemplar    = new data_impl    exemplar_ext in

let a1 = element_exemplar # cre-
ate_element dtd (T_element "a") ["att", "apple"]
and b1 = element_exemplar # create_element dtd (T_element "b") []
and c1 = element_exemplar # create_element dtd (T_element "c") []
and a2 = element_exemplar # cre-
ate_element dtd (T_element "a") ["att", "orange"]
in

let cherries = data_exemplar # create_data dtd "Cherries" in
let orange   = data_exemplar # create_data dtd "An orange" in

a1 # add_node b1;
a1 # add_node c1;
b1 # add_node a2;
b1 # add_node cherries;
a2 # add_node orange;
```

Alternatively, the last block of statements could also be written as:

```
a1 # set_nodes [b1; c1];
```

```
b1 # set_nodes [a2; cherries];
a2 # set_nodes [orange];
```

The root of the tree is `a1`, i.e. it is true that

```
x # root == a1
```

for every x from { `a1`, `a2`, `b1`, `c1`, `cherries`, `orange` }.

Furthermore, the following properties hold:

```
  a1 # attribute "att" = Value "apple"
& a2 # attribute "att" = Value "orange"

& cherries # data = "Cherries"
&   orange # data = "An orange"
&       a1 # data = "CherriesAn orange"

&       a1 # node_type = T_element "a"
&       a2 # node_type = T_element "a"
&       b1 # node_type = T_element "b"
&       c1 # node_type = T_element "c"
& cherries # node_type = T_data
&   orange # node_type = T_data

&       a1 # sub_nodes = [ b1; c1 ]
&       a2 # sub_nodes = [ orange ]
&       b1 # sub_nodes = [ a2; cherries ]
&       c1 # sub_nodes = []
& cherries # sub_nodes = []
&   orange # sub_nodes = []

&       a2 # parent == a1
&       b1 # parent == b1
&       c1 # parent == a1
& cherries # parent == b1
&   orange # parent == a2
```

Searching nodes. The following function searches all nodes of a tree for which a certain condition holds:

```
let rec search p t =
  if p t then
    t :: search_list p (t # sub_nodes)
  else
    search_list p (t # sub_nodes)
```

```
and search_list p l =
  match l with
    []      -> []
  | t :: l' -> (search p t) @ (search_list p l')
;;
```

For example, if you want to search all elements of a certain type `et`, the function `search` can be applied as follows:

```
let search_element_type et t =
  search (fun x -> x # node_type = T_element et) t
;;
```

Getting attribute values. Suppose we have the declaration:

```
<!ATTLIST e a CDATA #REQUIRED
            b CDATA #IMPLIED
            c CDATA "12345">
```

In this case, every element `e` must have an attribute `a`, otherwise the parser would indicate an error. If the O'Caml variable `n` holds the node of the tree corresponding to the element, you can get the value of the attribute `a` by

```
let value_of_a = n # required_string_attribute "a"
```

which is more or less an abbreviation for

```
let value_of_a =
  match n # attribute "a" with
    Value s -> s
  | _       -> assert false
```

- as the attribute is required, the `attribute` method always returns a `Value`.

In contrast to this, the attribute `b` can be omitted. In this case, the method `required_string_attribute` works only if the attribute is there, and the method will fail if the attribute is missing. To get the value, you can apply the method `optional_string_attribute`:

```
let value_of_b = n # optional_string_attribute "b"
```

Now, `value_of_b` is of type `string option`, and `None` represents the omitted attribute. Alternatively, you could also use `attribute`:

```
let value_of_b =
```

```
  match n # attribute "b" with
    Value s       -> Some s
  | Implied_value -> None
  | _             -> assert false
```

The attribute `c` behaves much like `a`, because it has always a value. If the attribute is omitted, the default, here "12345", will be returned instead. Because of this, you can again use `required_string_attribute` to get the value.

The type `CDATA` is the most general string type. The types `NMTOKEN`, `ID`, `IDREF`, `ENTITY`, and all enumerators and notations are special forms of string types that restrict the possible values. From O'Caml, they behave like `CDATA`, i.e. you can use the methods `required_string_attribute` and `optional_string_attribute`, too.

In contrast to this, the types `NMTOKENS`, `IDREFS`, and `ENTITIES` mean lists of strings. Suppose we have the declaration:

```
<!ATTLIST f d NMTOKENS #REQUIRED
            e NMTOKENS #IMPLIED>
```

The type `NMTOKENS` stands for lists of space-separated tokens; for example the value `"1 abc 23ef"` means the list `["1"; "abc"; "23ef"]`. (Again, `IDREFS` and `ENTITIES` have more restricted values.) To get the value of attribute `d`, one can use

```
let value_of_d = n # required_list_attribute "d"
```

or

```
let value_of_d =
  match n # attribute "d" with
    Valuelist l -> l
  | _           -> assert false
```

As `d` is required, the attribute cannot be omitted, and the `attribute` method returns always a `Valuelist`.

For optional attributes like `e`, apply

```
let value_of_e = n # optional_list_attribute "e"
```

or

```
let value_of_e =
  match n # attribute "e" with
    Valuelist l   -> l
  | Implied_value -> []
```

```
  | _              -> assert false
```

Here, the case that the attribute is missing counts like the empty list.

3.2.7. Iterators

There are also several iterators in Pxp_document; please see the mli file for details. You can find examples for them in the "simple_transformation" directory.

```
val find : ?deeply:bool ->
           f:('ext node -> bool) -> 'ext node -> 'ext node

val find_all : ?deeply:bool ->
               f:('ext node -> bool) -> 'ext node -> 'ext node list

val find_element : ?deeply:bool ->
                   string -> 'ext node -> 'ext node

val find_all_elements : ?deeply:bool ->
                        string -> 'ext node -> 'ext node list

exception Skip
val map_tree :  pre:('exta node -> 'extb node) ->
               ?post:('extb node -> 'extb node) ->
               'exta node ->
                   'extb node


val map_tree_sibl :
        pre: ('exta node option -> 'exta node -> 'exta node option ->
                  'extb node) ->
       ?post:('extb node option -> 'extb node -> 'extb node option ->
                  'extb node) ->
       'exta node ->
           'extb node

val iter_tree : ?pre:('ext node -> unit) ->
                ?post:('ext node -> unit) ->
                'ext node ->
                    unit

val iter_tree_sibl :
       ?pre: ('ext node option -> 'ext node -> 'ext node option -> unit) ->
```

```
?post:('ext node option -> 'ext node -> 'ext node option -> unit) ->
'ext node ->
    unit
```

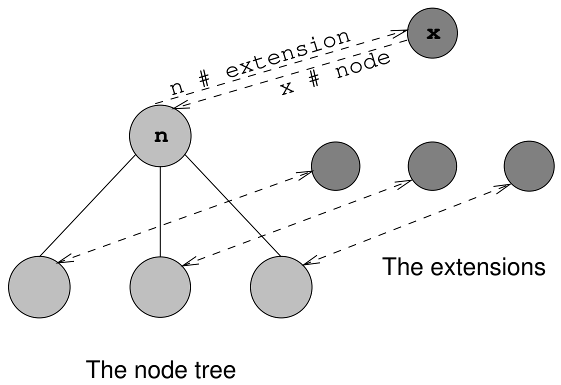
3.3. The class type `extension`

```
class type [ 'node ] extension =
  object ('self)
    method clone : 'self
      (* "clone" should return an exact deep copy of the object. *)
    method node : 'node
      (* "node" returns the corresponding node of this extension. This method
       * intended to return exactly what previ-
ously has been set by "set_node".
       *)
    method set_node : 'node -> unit
      (* "set_node" is invoked once the extension is associated to a new
       * node object.
       *)
  end
```

This is the type of classes used for node extensions. For every node of the document tree, there is not only the `node` object, but also an `extension` object. The latter has minimal functionality; it has only the necessary methods to be attached to the node object containing the details of the node instance. The extension object is called extension because its purpose is extensibility.

For some reasons, it is impossible to derive the `node` classes (i.e. `element_impl` and `data_impl`) such that the subclasses can be extended by new new methods. But subclassing nodes is a great feature, because it allows the user to provide different classes for different types of nodes. The extension objects are a workaround that is as powerful as direct subclassing, the costs are some notation overhead.

Figure 3-6. The structure of nodes and extensions



The picture shows how the nodes and extensions are linked together. Every node has a reference to its extension, and every extension has a reference to its node. The methods `extension` and `node` follow these references; a typical phrase is

```
self # node # attribute "xy"
```

to get the value of an attribute from a method defined in the extension object; or

```
self # node # iter
  (fun n -> n # extension # my_method ...)
```

to iterate over the subnodes and to call `my_method` of the corresponding extension objects.

Note that extension objects do not have references to subnodes (or "subextensions") themselves; in order to get one of the children of an extension you must first go to the node object, then get the child node, and finally reach the extension that is logically the child of the extension you started with.

3.3.1. How to define an extension class

At minimum, you must define the methods `clone`, `node`, and `set_node` such that your class is compatible with the type `extension`. The method `set_node` is called during the initialization of the node, or after a node has been cloned; the node object invokes `set_node` on the extension object to tell it that this node is now the object the extension is linked to. The extension must return the node object passed as argument of `set_node` when the `node` method is called.

The `clone` method must return a copy of the extension object; at least the object itself must be duplicated, but if required, the copy should deeply duplicate all objects and values that are referred by the extension, too. Whether this is required, depends on the application; `clone` is invoked by the node object when one of its cloning methods is called.

A good starting point for an extension class:

```
class custom_extension =
  object (self)

    val mutable node = (None : custom_extension node option)

    method clone = {< >}

    method node =
      match node with
          None ->
            assert false
        | Some n -> n

    method set_node n =
      node <- Some n

  end
```

This class is compatible with `extension`. The purpose of defining such a class is, of course, adding further methods; and you can do it without restriction.

Often, you want not only one extension class. In this case, it is the simplest way that all your classes (for one kind of document) have the same type (with respect to the interface; i.e. it does not matter if your classes differ in the defined private methods and instance variables, but public methods count). This approach avoids lots of coercions and problems with type incompatibilities. It is simple to implement:

```
class custom_extension =
  object (self)
    val mutable node = (None : custom_extension node option)

    method clone = ...       (* see above *)
    method node = ...        (* see above *)
    method set_node n = ... (* see above *)

    method virtual my_method1 : ...
    method virtual my_method2 : ...
    ... (* etc. *)
  end
```

```
class custom_extension_kind_A =
  object (self)
    inherit custom_extension

    method my_method1 = ...
    method my_method2 = ...
  end

class custom_extension_kind_B =
  object (self)
    inherit custom_extension

    method my_method1 = ...
    method my_method2 = ...
  end
```

If a class does not need a method (e.g. because it does not make sense, or it would violate some important condition), it is possible to define the method and to always raise an exception when the method is invoked (e.g. `assert false`).

The latter is a strong recommendation: do not try to further specialize the types of extension objects. It is difficult, sometimes even impossible, and almost never worth-while.

3.3.2. How to bind extension classes to element types

Once you have defined your extension classes, you can bind them to element types. The simplest case is that you have only one class and that this class is to be always used. The parsing functions in the module `Pxp_yacc` take a `spec` argument which can be customized. If your single class has the name `c`, this argument should be

```
let spec =
  make_spec_from_alist
    ~data_exemplar:            (new data_impl c)
    ~default_element_exemplar: (new element_impl c)
    ~element_alist:            []
    ()
```

This means that data nodes will be created from the exemplar passed by ~data_exemplar and that all element nodes will be made from the exemplar specified by ~default_element_exemplar. In ~element_alist, you can pass that different exemplars are to be used for different element types; but this is an optional feature. If you do not need it, pass the empty list.

Remember that an exemplar is a (node, extension) pair that serves as pattern when new nodes (and the corresponding extension objects) are added to the document tree. In this case, the exemplar contains `c` as extension, and when nodes are created, the exemplar is cloned, and cloning makes also a copy of `c` such that all nodes of the document tree will have a copy of `c` as extension.

The `~element_alist` argument can bind specific element types to specific exemplars; as exemplars may be instances of different classes it is effectively possible to bind element types to classes. For example, if the element type "p" is implemented by class "c_p", and "q" is realized by "c_q", you can pass the following value:

```
let spec =
  make_spec_from_alist
    ~data_exemplar:            (new data_impl c)
    ~default_element_exemplar: (new element_impl c)
    ~element_alist:
      [ "p", new element_impl c_p;
        "q", new element_impl c_q;
      ]
    ()
```

The extension object `c` is still used for all data nodes and for all other element types.

3.4. Details of the mapping from XML text to the tree representation

3.4.1. The representation of character-free elements

If an element declaration does not allow the element to contain character data, the following rules apply.

If the element must be empty, i.e. it is declared with the keyword `EMPTY`, the element instance must be effectively empty (it must not even contain whitespace characters). The parser guarantees that a declared `EMPTY` element does never contain a data node, even if the data node represents the empty string.

If the element declaration only permits other elements to occur within that element but not character data, it is still possible to insert whitespace characters between the subelements. The parser ignores these characters, too, and does not create data nodes for them.

Example. Consider the following element types:

```
<!ELEMENT x ( #PCDATA | z )* >
<!ELEMENT y ( z )* >
```

```
<!ELEMENT z EMPTY>
```

Only `x` may contain character data, the keyword `#PCDATA` indicates this. The other types are character-free.

The XML term

```
<x><z/> <z/></x>
```

will be internally represented by an element node for `x` with three subnodes: the first `z` element, a data node containing the space character, and the second `z` element. In contrast to this, the term

```
<y><z/> <z/></y>
```

is represented by an element node for `y` with only *two* subnodes, the two `z` elements. There is no data node for the space character because spaces are ignored in the character-free element `y`.

3.4.2. The representation of character data

The XML specification allows all Unicode characters in XML texts. This parser can be configured such that UTF-8 is used to represent the characters internally; however, the default character encoding is ISO-8859-1. (Currently, no other encodings are possible for the internal string representation; the type `Pxp_types.rep_encoding` enumerates the possible encodings. Principially, the parser could use any encoding that is ASCII-compatible, but there are currently only lexical analyzers for UTF-8 and ISO-8859-1. It is currently impossible to use UTF-16 or UCS-4 as internal encodings (or other multibyte encodings which are not ASCII-compatible) unless major parts of the parser are rewritten - unlikely...)

The internal encoding may be different from the external encoding (specified in the XML declaration `<?xml ... encoding="..."?>`); in this case the strings are automatically converted to the internal encoding.

If the internal encoding is ISO-8859-1, it is possible that there are characters that cannot be represented. In this case, the parser ignores such characters and prints a warning (to the `collect_warning` object that must be passed when the parser is called).

The XML specification allows lines to be separated by single LF characters, by CR LF character sequences, or by single CR characters. Internally, these separators are always converted to single LF characters.

The parser guarantees that there are never two adjacent data nodes; if necessary, data material that would otherwise be represented by several nodes is collapsed into one node. Note that you can still create node trees with adjacent data nodes; however, the parser does not return such trees.

Note that CDATA sections are not represented specially; such sections are added to the current data material that being collected for the next data node.

3.4.3. The representation of entities within documents

Entities are not represented within documents! If the parser finds an entity reference in the document content, the reference is immediately expanded, and the parser reads the expansion text instead of the reference.

3.4.4. The representation of attributes

As attribute values are composed of Unicode characters, too, the same problems with the character encoding arise as for character material. Attribute values are converted to the internal encoding, too; and if there are characters that cannot be represented, these are dropped, and a warning is printed.

Attribute values are normalized before they are returned by methods like `attribute`. First, any remaining entity references are expanded; if necessary, expansion is performed recursively. Second, newline characters (any of LF, CR LF, or CR characters) are converted to single space characters. Note that especially the latter action is prescribed by the XML standard (but is not converted such that it is still possible to include line feeds into attributes).

3.4.5. The representation of processing instructions

Processing instructions are parsed to some extent: The first word of the PI is called the target, and it is stored separated from the rest of the PI:

```
<?target rest?>
```

The exact location where a PI occurs is not represented (by default). The parser puts the PI into the object that represents the embracing construct (an element, a DTD, or the whole document); that means you can find out which PIs occur in a certain element, in the DTD, or in the whole document, but you cannot lookup the exact position within the construct.

If you require the exact location of PIs, it is possible to create extra nodes for them. This mode is controled by the option `enable_pinstr_nodes`. The additional nodes have the node type `T_pinstr` *`target`*, and are created from special exemplars contained in the `spec` (see pxp_document.mli).

3.4.6. The representation of comments

Normally, comments are not represented; they are dropped by default. However, if you require them, it is possible to create `T_comment` nodes for them. This mode can be specified by the option `enable_comment_nodes`. Comment nodes are created from special exemplars contained in the `spec` (see pxp_document.mli). You can access the contents of comments through the method `comment`.

3.4.7. The attributes `xml:lang` and `xml:space`

These attributes are not supported specially; they are handled like any other attribute.

3.4.8. And what about namespaces?

Currently, there is no special support for namespaces. However, the parser allows it that the colon occurs in names such that it is possible to implement namespaces on top of the current API.

Some future release of PXP will support namespaces as built-in feature...

Chapter 4. Configuring and calling the parser

4.1. Overview

There are the following main functions invoking the parser (in Pxp_yacc):

- *parse_document_entity:* You want to parse a complete and closed document consisting of a DTD and the document body; the body is validated against the DTD. This mode is interesting if you have a file

  ```
  <!DOCTYPE root ... [ ... ] > <root> ... </root>
  ```

and you can accept any DTD that is included in the file (e.g. because the file is under your control).

- *parse_wfdocument_entity:* You want to parse a complete and closed document consisting of a DTD and the document body; but the body is not validated, only checked for well-formedness. This mode is preferred if validation costs too much time or if the DTD is missing.
- *parse_dtd_entity:* You want only to parse an entity (file) containing the external subset of a DTD. Sometimes it is interesting to read such a DTD, for example to compare it with the DTD included in a document, or to apply the next mode:
- *parse_content_entity:* You want only to parse an entity (file) containing a fragment of a document body; this fragment is validated against the DTD you pass to the function. Especially, the fragment must not have a `<!DOCTYPE>` clause, and must directly begin with an element. The element is validated against the DTD. This mode is interesting if you want to check documents against a fixed, immutable DTD.
- *parse_wfcontent_entity:* This function also parses a single element without DTD, but does not validate it.
- *extract_dtd_from_document_entity:* This function extracts the DTD from a closed document consisting of a DTD and a document body. Both the internal and the external subsets are extracted.

In many cases, `parse_document_entity` is the preferred mode to parse a document in a validating way, and `parse_wfdocument_entity` is the mode of choice to parse a file while only checking for well-formedness.

There are a number of variations of these modes. One important application of a parser is to check documents of an untrusted source against a fixed DTD. One solution is to not allow the `<!DOCTYPE>` clause in these documents, and treat the document like a fragment (using mode *parse_content_entity*). This is very simple, but inflexible; users of such a system cannot even define additional entities to abbreviate frequent phrases of their text.

It may be necessary to have a more intelligent checker. For example, it is also possible to parse the document to check fully, i.e. with DTD, and to compare this DTD with the prescribed one. In order to fully parse the document, mode *parse_document_entity* is applied, and to get the DTD to compare with mode *parse_dtd_entity* can be used.

There is another very important configurable aspect of the parser: the so-called resolver. The task of the resolver is to locate the contents of an (external) entity for a given entity name, and to make the contents accessible as a character stream. (Furthermore, it also normalizes the character set; but this is a detail we can ignore here.) Consider you have a file called `"main.xml"` containing

```
<!ENTITY % sub SYSTEM "sub/sub.xml">
%sub;
```

and a file stored in the subdirectory `"sub"` with name `"sub.xml"` containing

```
<!ENTITY % subsub SYSTEM "subsub/subsub.xml">
%subsub;
```

and a file stored in the subdirectory `"subsub"` of `"sub"` with name `"subsub.xml"` (the contents of this file do not matter). Here, the resolver must track that the second entity `subsub` is located in the directory `"sub/subsub"`, i.e. the difficulty is to interpret the system (file) names of entities relative to the entities containing them, even if the entities are deeply nested.

There is not a fixed resolver already doing everything right - resolving entity names is a task that highly depends on the environment. The XML specification only demands that `SYSTEM` entities are interpreted like URLs (which is not very precise, as there are lots of URL schemes in use), hoping that this helps overcoming the local peculiarities of the environment; the idea is that if you do not know your environment you can refer to other entities by denoting URLs for them. I think that this interpretation of `SYSTEM` names may have some applications in the internet, but it is not the first choice in general. Because of this, the resolver is a separate module of the parser that can be exchanged by another one if necessary; more precisely, the parser already defines several resolvers.

The following resolvers do already exist:

- Resolvers reading from arbitrary input channels. These can be configured such that a certain ID is associated with the channel; in this case inner references to external entities can be resolved. There is also a special resolver that interprets SYSTEM IDs as URLs; this resolver can process relative SYSTEM names and determine the corresponding absolute URL.
- A resolver that reads always from a given O'Caml string. This resolver is not able to resolve further names unless the string is not associated with any name, i.e. if the document contained in the string refers to an external entity, this reference cannot be followed in this case.
- A resolver for file names. The `SYSTEM` name is interpreted as file URL with the slash "/" as separator for directories. - This resolver is derived from the generic URL resolver.

The interface a resolver must have is documented, so it is possible to write your own resolver. For example, you could connect the parser with an HTTP client, and resolve URLs of the HTTP namespace.

The resolver classes support that several independent resolvers are combined to one more powerful resolver; thus it is possible to combine a self-written resolver with the already existing resolvers.

Note that the existing resolvers only interpret `SYSTEM` names, not `PUBLIC` names. If it helps you, it is possible to define resolvers for `PUBLIC` names, too; for example, such a resolver could look up the public name in a hash table, and map it to a system name which is passed over to the existing resolver for system names. It is relatively simple to provide such a resolver.

4.2. Resolvers and sources

4.2.1. Using the built-in resolvers (called sources)

The type `source` enumerates the two possibilities where the document to parse comes from.

```
type source =
    Entity of ((dtd -> Pxp_entity.entity) * Pxp_reader.resolver)
  | ExtID of (ext_id * Pxp_reader.resolver)
```

You normally need not to worry about this type as there are convenience functions that create `source` values:

- `from_file s`: The document is read from file `s`; you may specify absolute or relative path names. The file name must be encoded as UTF-8 string.

  There is an optional argument `~system_encoding` specifying the character encoding which is used for the names of the file system. For example, if this encoding is ISO-8859-1 and `s` is also a ISO-8859-1 string, you can form the source:

  ```
  let s_utf8  =  recode_string ~in_enc:`Enc_iso88591 ~out_enc:`Enc_utf8 s in
  from_file ~system_encoding:`Enc_iso88591 s_utf8
  ```

  This `source` has the advantage that it is able to resolve inner external entities; i.e. if your document includes data from another file (using the `SYSTEM` attribute), this mode will find that file. However, this mode cannot resolve `PUBLIC` identifiers nor `SYSTEM` identifiers other than "file:".

- `from_channel ch`: The document is read from the channel `ch`. In general, this source also supports file URLs found in the document; however, by default only absolute URLs are understood. It is possible to associate an ID with the channel such that the resolver knows how to interpret relative URLs:

  ```
  from_channel ~id:(System "file:///dir/dir1/") ch
  ```

There is also the ~system_encoding argument specifying how file names are encoded. - The example from above can also be written (but it is no longer possible to interpret relative URLs because there is no ~id argument, and computing this argument is relatively complicated because it must be a valid URL):

```
let ch = open_in s in
let src = from_channel ~system_encoding:`Enc_iso88591 ch in
...;
close_in ch
```

- `from_string s`: The string `s` is the document to parse. This mode is not able to interpret file names of `SYSTEM` clauses, nor it can look up `PUBLIC` identifiers.

  Normally, the encoding of the string is detected as usual by analyzing the XML declaration, if any. However, it is also possible to specify the encoding directly:

  ```
  let src = from_string ~fixenc:`ISO-8859-2 s
  ```

- `ExtID (id, r)`: The document to parse is denoted by the identifier `id` (either a `SYSTEM` or `PUBLIC` clause), and this identifier is interpreted by the resolver `r`. Use this mode if you have written your own resolver.

  Which character sets are possible depends on the passed resolver `r`.

- `Entity (get_entity, r)`: The document to parse is returned by the function invocation `get_entity dtd`, where `dtd` is the DTD object to use (it may be empty). Inner external references occuring in this entity are resolved using the resolver `r`.

  Which character sets are possible depends on the passed resolver `r`.

4.2.2. The resolver API

A resolver is an object that can be opened like a file, but you do not pass the file name to the resolver, but the XML identifier of the entity to read from (either a `SYSTEM` or `PUBLIC` clause). When opened, the resolver must return the `Lexing.lexbuf` that reads the characters. The resolver can be closed, and it can be cloned. Furthermore, it is possible to tell the resolver which character set it should assume. - The following from Pxp_reader:

```
exception Not_competent
exception Not_resolvable of exn

class type resolver =
  object
    method init_rep_encoding : rep_encoding -> unit
    method init_warner : collect_warnings -> unit
```

```
    method rep_encoding : rep_encoding
    method open_in : ext_id -> Lexing.lexbuf
    method close_in : unit
    method change_encoding : string -> unit
    method clone : resolver
    method close_all : unit
  end
```

The resolver object must work as follows:

- When the parser is called, it tells the resolver the warner object and the internal encoding by invoking `init_warner` and `init_rep_encoding`. The resolver should store these values. The method `rep_encoding` should return the internal encoding.
- If the parser wants to read from the resolver, it invokes the method `open_in`. Either the resolver succeeds, in which case the `Lexing.lexbuf` reading from the file or stream must be returned, or opening fails. In the latter case the method implementation should raise an exception (see below).
- If the parser finishes reading, it calls the `close_in` method.
- If the parser finds a reference to another external entity in the input stream, it calls `clone` to get a second resolver which must be initially closed (not yet connected with an input stream). The parser then invokes `open_in` and the other methods as described.
- If you already know the character set of the input stream, you should recode it to the internal encoding, and define the method `change_encoding` as an empty method.
- If you want to support multiple external character sets, the object must follow a much more complicated protocol. Directly after `open_in` has been called, the resolver must return a lexical buffer that only reads one byte at a time. This is only possible if you create the lexical buffer with `Lexing.from_function`; the function must then always return 1 if the EOF is not yet reached, and 0 if EOF is reached. If the parser has read the first line of the document, it will invoke `change_encoding` to tell the resolver which character set to assume. From this moment, the object can return more than one byte at once. The argument of `change_encoding` is either the parameter of the "encoding" attribute of the XML declaration, or the empty string if there is not any XML declaration or if the declaration does not contain an encoding attribute.

  At the beginning the resolver must only return one character every time something is read from the lexical buffer. The reason for this is that you otherwise would not exactly know at which position in the input stream the character set changes.

  If you want automatic recognition of the character set, it is up to the resolver object to implement this.
- If an error occurs, the parser calls the method `close_all` for the top-level resolver; this method should close itself (if not already done) and all clones.

Exceptions. It is possible to chain resolvers such that when the first resolver is not able to open the entity, the other resolvers of the chain are tried in turn. The method `open_in` should raise the exception

Not_competent to indicate that the next resolver should try to open the entity. If the resolver is able to handle the ID, but some other error occurs, the exception Not_resolvable should be raised to force that the chain breaks.

Example: How to define a resolver that is equivalent to from_string: ...

4.2.3. Predefined resolver components

There are some classes in Pxp_reader that define common resolver behaviour.

```
class resolve_read_this_channel :
    ?id:ext_id ->
    ?fixenc:encoding ->
    ?auto_close:bool ->
    in_channel ->
        resolver
```

Reads from the passed channel (it may be even a pipe). If the ~id argument is passed to the object, the created resolver accepts only this ID. Otherwise all IDs are accepted. - Once the resolver has been cloned, it does not accept any ID. This means that this resolver cannot handle inner references to external entities. Note that you can combine this resolver with another resolver that can handle inner references (such as resolve_as_file); see class 'combine' below. - If you pass the ~fixenc argument, the encoding of the channel is set to the passed value, regardless of any auto-recognition or any XML declaration. - If ~auto_close = true (which is the default), the channel is closed after use. If ~auto_close = false, the channel is left open.

```
class resolve_read_any_channel :
    ?auto_close:bool ->
    channel_of_id:(ext_id -> (in_channel * encoding option)) ->
        resolver
```

This resolver calls the function ~channel_of_id to open a new channel for the passed ext_id. This function must either return the channel and the encoding, or it must fail with Not_competent. The function must return None as encoding if the default mechanism to recognize the encoding should be used. It must return Some e if it is already known that the encoding of the channel is e. If ~auto_close = true (which is the default), the channel is closed after use. If ~auto_close = false, the channel is left open.

```
class resolve_read_url_channel :
    ?base_url:Neturl.url ->
    ?auto_close:bool ->
    url_of_id:(ext_id -> Neturl.url) ->
```

```
    channel_of_url:(Neturl.url -> (in_channel * encoding option)) ->
        resolver
```

When this resolver gets an ID to read from, it calls the function `~url_of_id` to get the corresponding URL. This URL may be a relative URL; however, a URL scheme must be used which contains a path. The resolver converts the URL to an absolute URL if necessary. The second function, `~channel_of_url`, is fed with the absolute URL as input. This function opens the resource to read from, and returns the channel and the encoding of the resource.

Both functions, `~url_of_id` and `~channel_of_url`, can raise Not_competent to indicate that the object is not able to read from the specified resource. However, there is a difference: A Not_competent from `~url_of_id` is left as it is, but a Not_competent from `~channel_of_url` is converted to Not_resolvable. So only `~url_of_id` decides which URLs are accepted by the resolver and which not.

The function `~channel_of_url` must return `None` as encoding if the default mechanism to recognize the encoding should be used. It must return `Some e` if it is already known that the encoding of the channel is `e`.

If `~auto_close = true` (which is the default), the channel is closed after use. If `~auto_close = false`, the channel is left open.

Objects of this class contain a base URL relative to which relative URLs are interpreted. When creating a new object, you can specify the base URL by passing it as `~base_url` argument. When an existing object is cloned, the base URL of the clone is the URL of the original object. - Note that the term "base URL" has a strict definition in RFC 1808.

```
class resolve_read_this_string :
    ?id:ext_id ->
    ?fixenc:encoding ->
    string ->
        resolver
```

Reads from the passed string. If the `~id` argument is passed to the object, the created resolver accepts only this ID. Otherwise all IDs are accepted. - Once the resolver has been cloned, it does not accept any ID. This means that this resolver cannot handle inner references to external entities. Note that you can combine this resolver with another resolver that can handle inner references (such as resolve_as_file); see class 'combine' below. - If you pass the `~fixenc` argument, the encoding of the string is set to the passed value, regardless of any auto-recognition or any XML declaration.

```
class resolve_read_any_string :
    string_of_id:(ext_id -> (string * encoding option)) ->
        resolver
```

This resolver calls the function ~string_of_id to get the string for the passed ext_id. This function must either return the string and the encoding, or it must fail with Not_competent. The function must return None as encoding if the default mechanism to recognize the encoding should be used. It must return Some e if it is already known that the encoding of the string is e.

```
class resolve_as_file :
    ?file_prefix:[ ‘Not_recognized | ‘Allowed | ‘Required ] ->
    ?host_prefix:[ ‘Not_recognized | ‘Allowed | ‘Required ] ->
    ?system_encoding:encoding ->
    ?url_of_id:(ext_id -> Neturl.url) ->
    ?channel_of_url: (Neturl.url -> (in_channel * encoding option)) ->
    unit ->
        resolver
```

Reads from the local file system. Every file name is interpreted as file name of the local file system, and the referred file is read.

The full form of a file URL is: file://host/path, where ’host’ specifies the host system where the file identified ’path’ resides. host = "" or host = "localhost" are accepted; other values will raise Not_competent. The standard for file URLs is defined in RFC 1738.

Option ~file_prefix: Specifies how the "file:" prefix of file names is handled:

- ‘Not_recognized:The prefix is not recognized.
- ‘Allowed: The prefix is allowed but not required (the default).
- ‘Required: The prefix is required.

Option ~host_prefix: Specifies how the "//host" phrase of file names is handled:

- ‘Not_recognized:The prefix is not recognized.
- ‘Allowed: The prefix is allowed but not required (the default).
- ‘Required: The prefix is required.

Option ~system_encoding: Specifies the encoding of file names of the local file system. Default: UTF-8.

Options ~url_of_id, ~channel_of_url: Not for the casual user!

```
class combine :
    ?prefer:resolver ->
    resolver list ->
        resolver
```

Combines several resolver objects. If a concrete entity with an `ext_id` is to be opened, the combined resolver tries the contained resolvers in turn until a resolver accepts opening the entity (i.e. it does not raise Not_competent on open_in).

Clones: If the 'clone' method is invoked before 'open_in', all contained resolvers are cloned separately and again combined. If the 'clone' method is invoked after 'open_in' (i.e. while the resolver is open), additionally the clone of the active resolver is flagged as being preferred, i.e. it is tried first.

4.3. The DTD classes

Sorry, not yet written. Perhaps the interface definition of Pxp_dtd expresses the same:

```
(**********************************************************************)
(*                                                                    *)
(* Pxp_dtd:                                                           *)
(*     Object model of document type declarations                     *)
(*                                                                    *)
(**********************************************************************)

(* ======================================================================
 * OVERVIEW
 *
 * class dtd ............... represents the whole DTD, including element
 *                           declarations, entity declarations, notation
 *                           declarations, and processing instructions
 * class dtd_element ....... represents an element declaration consisting
 *                           of a content model and an attribute list
 *                           declaration
 * class dtd_notation ...... represents a notation declaration
 * class proc_instruction .. represents a processing instruction
 * ======================================================================
 *
 *)


class dtd :
  (* Creation:
   *   new dtd
   * creates a new, empty DTD object without any declaration, without a root
   * element, without an ID.
   *)
```

```
  Pxp_types.collect_warnings ->
  Pxp_types.rep_encoding ->
  object
    method root : string option
      (* get the name of the root element if present *)

    method set_root : string -> unit
      (* set the name of the root element. This method can be invoked
       * only once
       *)

    method id : Pxp_types.dtd_id option
      (* get the identifier for this DTD *)

    method set_id : Pxp_types.dtd_id -> unit
      (* set the identifier. This method can be invoked only once *)

    method encoding : Pxp_types.rep_encoding
      (* returns the encoding used for character representation *)


    method allow_arbitrary : unit
      (* After this method has been invoked, the ob-
ject changes its behaviour:
       * - elements and notations that have not been added may be used in an
       *   arbitrary way; the methods "element" and "notation" indicate this
       *   by raising Undeclared instead of Validation_error.
       *)

    method disallow_arbitrary : unit

    method arbitrary_allowed : bool
      (* Returns whether arbitrary contents are allowed or not. *)

    method standalone_declaration : bool
      (* Whether there is a 'standalone' declaration or not. Strictly
       * speaking, this declaration is not part of the DTD, but it is
       * included here because of practical reasons.
       * If not set, this property defaults to 'false'.
       *)

    method set_standalone_declaration : bool -> unit
      (* Sets the 'standalone' declaration. *)
```

```
    method add_element : dtd_element -> unit
      (* add the given element declaration to this DTD. Raises Not_found
       * if there is already an element declaration with the same name.
       *)

    method add_gen_entity : Pxp_entity.entity -> bool -> unit
      (* add_gen_entity e extdecl:
       * add the entity 'e' as general entity to this DTD (general entities
       * are those represented by &name;). If there is already a declaration
       * with the same name, the second definition is ignored; as excep-
tion from
       * this rule, entities with names "lt", "gt", "amp", "quot", and "apos"
       * may only be redeclared with a definition that is equivalent to the
       * standard definition; otherwise a Validation_error is raised.
       *
       * 'extdecl': 'true' indicates that the entity declaration occurs in
       * an external entity. (Used for the standalone check.)
       *)

    method add_par_entity : Pxp_entity.entity -> unit
      (* add the given entity as parameter entity to this DTD (parameter
       * entities are those represented by %name;). If there is already a
       * declaration with the same name, the second definition is ignored.
       *)

    method add_notation : dtd_notation -> unit
      (* add the given notation to this DTD. If there is al-
ready a declaration
       * with the same name, a Validation_error is raised.
       *)

    method add_pinstr : proc_instruction -> unit
      (* add the given processing instruction to this DTD. *)

    method element : string -> dtd_element
      (* looks up the element declaration with the given name. Raises
       * Validation_error if the element can-
not be found. (If "allow_arbitrary"
       * has been invoked before, Unrestricted is raised instead.)
       *)

    method element_names : string list
      (* returns the list of the names of all element declarations. *)

    method gen_entity : string -> (Pxp_entity.entity * bool)
```

```
      (* let e, extdecl = obj # gen_entity n:
       * looks up the general entity 'e' with the name 'n'. Raises
       * WF_error if the entity cannot be found.
       * 'extdecl': indicates whether the entity declaration occured in an
       * external entity.
       *)

    method gen_entity_names : string list
      (* returns the list of all general entity names *)

    method par_entity : string -> Pxp_entity.entity
      (* looks up the parameter entity with the given name. Raises
       * WF_error if the entity cannot be found.
       *)

    method par_entity_names : string list
      (* returns the list of all parameter entity names *)

    method notation : string -> dtd_notation
      (* looks up the notation declaration with the given name. Raises
       * Validation_error if the notation can-
not be found. (If "allow_arbitrary"
       * has been invoked before, Unrestricted is raised instead.)
       *)

    method notation_names : string list
      (* Returns the list of the names of all added notations *)

    method pinstr : string -> proc_instruction list
      (* looks up all processing instructions with the given target.
       * The "target" is the identifier following "<?".
       * Note: It is not possible to find out the exact position of the
       * processing instruction.
       *)

    method pinstr_names : string list
      (* Returns the list of the names (targets) of all added pinstrs *)

    method validate : unit
      (* ensures that the DTD is valid. This method is optimized such that
       * actual validation is only performed if DTD has changed.
       * If the DTD is invalid, mostly a Validation_error is raised,
       * but other exceptions are possible, too.
       *)
```

```
    method only_deterministic_models : unit
      (* Succeeds if all regexp content models are deterministic.
       * Otherwise Validation_error.
       *)

    method write : Pxp_types.output_stream -> Pxp_types.encoding -> bool -
> unit
      (* write_compact_as_latin1 os enc doctype:
       * Writes the DTD as 'enc'-encoded string to 'os'. If 'doctype', a
       * DTD like <!DOCTYPE root [ ... ]> is written. If 'not doctype',
       * only the declarations are written (the material within the
       * square brackets).
       *)

    method write_compact_as_latin1 : Pxp_types.output_stream -> bool -> unit
      (* DEPRECATED METHOD; included only to keep compatibility with
       * older versions of the parser
       *)


    (*----------------------------*)
    method invalidate : unit
      (* INTERNAL METHOD *)
    method warner : Pxp_types.collect_warnings
      (* INTERNAL METHOD *)
  end



(* ---------------------------------------------- *)

and dtd_element : dtd -> string ->
  (* Creation:
   *   new dtd_element init_dtd init_name:
   * creates a new dtd_element object for init_dtd with init_name.
   * The strings are represented in the same encoding as init_dtd.
   *)
  object

    method name : string
      (* returns the name of the declared element *)

    method externally_declared : bool
      (* returns whether the element declaration occurs in an external
       * entity.
```

```
     *)

  method content_model : Pxp_types.content_model_type
    (* get the content model of this element declaration, or Unspecified *)

  method content_dfa : Pxp_dfa.dfa_definition option
    (* return the DFA of the content model if there is a DFA, or None.
     * A DFA exists only for regexp style content models which are
     * deterministic.
     *)

  method set_cm_and_extdecl : Pxp_types.content_model_type -> bool -> unit
    (* set_cm_and_extdecl cm extdecl:
     * set the content model to 'cm'. Once the content model is not
     * Unspecified, it cannot be set to a different value again.
     * Furthermore, it is set whether the element occurs in an external
     * entity ('extdecl').
     *)

  method encoding : Pxp_types.rep_encoding
    (* Return the encoding of the strings *)

  method allow_arbitrary : unit
    (* After this method has been invoked, the ob-
ject changes its behaviour:
     * - attributes that have not been added may be used in an
     *   arbitrary way; the method "attribute" indicates this
     *   by raising Undeclared instead of Validation_error.
     *)

  method disallow_arbitrary : unit

  method arbitrary_allowed : bool
    (* Returns whether arbitrary attributes are allowed or not. *)

  method attribute : string ->
                       Pxp_types.att_type * Pxp_types.att_default
    (* get the type and default value of a declared attribute, or raise
     * Validation_error if the attribute does not exist.
     * If 'arbitrary_allowed', the exception Undeclared is raised instead
     * of Validation_error.
     *)

  method attribute_violates_standalone_declaration :
            string -> string option -> bool
```

```
      (* attribute_violates_standalone_declaration name v:
       * Checks whether the attribute 'name' violates the "standalone"
       * declaration if it has value 'v'.
       * The method returns true if:
       * - The attribute declaration occurs in an external entity,
       * and if one of the two conditions holds:
       * - v = None, and there is a default for the attribute value
       * - v = Some s, and the type of the attribute is not CDATA,
       *   and s changes if normalized according to the rules of the
       *   attribute type.
       *
       * The method raises Validation_error if the attribute does not exist.
       * If 'arbitrary_allowed', the exception Undeclared is raised instead
       * of Validation_error.
       *)

    method attribute_names : string list
      (* get the list of all declared attributes *)

    method names_of_required_attributes : string list
      (* get the list of all attributes that are specified as required
       * attributes
       *)

    method id_attribute_name : string option
      (* Returns the name of the attribute with type ID, or None. *)

    method idref_attribute_names : string list
      (* Returns the names of the attributes with type IDREF or IDREFS. *)

    method add_attribute : string ->
                           Pxp_types.att_type ->
   Pxp_types.att_default ->
   bool ->
     unit
      (* add_attribute name type default extdecl:
       * add an attribute declaration for an attribute with the given name,
       * type, and default value. If there is more than one declaration for
       * an attribute name, the first declara-
tion counts; the other declarations
       * are ignored.
       * 'extdecl': if true, the attribute declaration occurs in an external
       * entity. This property is used to check the "standalone" attribute.
       *)
```

```
    method validate : unit
      (* checks whether this element declaration (i.e. the content model and
       * all attribute declarations) is valid for the associated DTD.
       * Raises mostly Validation_error if the validation fails.
       *)

    method write : Pxp_types.output_stream -> Pxp_types.encoding -> unit
      (* write_compact_as_latin1 os enc:
       * Writes the <!ELEMENT ... > declaration to 'os' as 'enc'-
encoded string.
       *)

    method write_compact_as_latin1 : Pxp_types.output_stream -> unit
      (* DEPRECATED METHOD; included only to keep compatibility with
       * older versions of the parser
       *)
  end

(* ----------------------------------------------- *)

and dtd_notation : string -> Pxp_types.ext_id -> Pxp_types.rep_encoding ->
  (* Creation:
   *    new dtd_notation a_name an_external_ID init_encoding
   * creates a new dtd_notation object with the given name and the given
   * external ID.
   *)
  object
    method name : string
    method ext_id : Pxp_types.ext_id
    method encoding : Pxp_types.rep_encoding

    method write : Pxp_types.output_stream -> Pxp_types.encoding -> unit
      (* write_compact_as_latin1 os enc:
       * Writes the <!NOTATION ... > declaration to 'os' as 'enc'-encoded
       * string.
       *)

    method write_compact_as_latin1 : Pxp_types.output_stream -> unit
      (* DEPRECATED METHOD; included only to keep compatibility with
       * older versions of the parser
       *)

  end

(* ----------------------------------------------- *)
```

```
and proc_instruction : string -> string -> Pxp_types.rep_encoding ->
  (* Creation:
   *   new proc_instruction a_target a_value
   * creates a new proc_instruction object with the given target string and
   * the given value string.
   * Note: A processing instruction is written as <?target value?>.
   *)
  object
    method target : string
    method value : string
    method encoding : Pxp_types.rep_encoding

    method write : Pxp_types.output_stream -> Pxp_types.encoding -> unit
      (* write os enc:
       * Writes the <?...?> PI to 'os' as 'enc'-encoded string.
       *)

    method write_compact_as_latin1 : Pxp_types.output_stream -> unit
      (* DEPRECATED METHOD; included only to keep compatibility with
       * older versions of the parser
       *)

    method parse_pxp_option : (string * string * (string * string) list)
      (* Parses a PI containing a PXP option. Such PIs are formed like:
       *   <?target option-name option-att="value" option-att="value" ... ?>
       * The method returns a triple
       *   (target, option-name, [option-att, value; ...])
       * or raises Error.
       *)

  end

;;
```

4.4. Invoking the parser

Here a description of Pxp_yacc.

4.4.1. Defaults

The following defaults are available:

```
val default_config : config
val default_extension : ('a node extension) as 'a
val default_spec : ('a node extension as 'a) spec
```

4.4.2. Parsing functions

In the following, the term "closed document" refers to an XML structure like

```
<!DOCTYPE ... [ declarations ] >
<root>
...
</root>
```

The term "fragment" refers to an XML structure like

```
<root>
...
</root>
```

i.e. only to one isolated element instance.

```
val parse_dtd_entity : config -> source -> dtd
```

Parses the declarations which are contained in the entity, and returns them as `dtd` object.

```
val extract_dtd_from_document_entity : config -> source -> dtd
```

Extracts the DTD from a closed document. Both the internal and the external subsets are extracted and combined to one `dtd` object. This function does not parse the whole document, but only the parts that are necessary to extract the DTD.

```
val parse_document_entity :
    ?transform_dtd:(dtd -> dtd) ->
    ?id_index:('ext index) ->
    config ->
    source ->
    'ext spec ->
        'ext document
```

Parses a closed document and validates it against the DTD that is contained in the document (internal and external subsets). The option `~transform_dtd` can be used to transform the DTD in the document, and to use the transformed DTD for validation. If `~id_index` is specified, an index of all ID attributes is created.

```
val parse_wfdocument_entity :
    config ->
    source ->
    'ext spec ->
        'ext document
```

Parses a closed document, but checks it only on well-formedness.

```
val parse_content_entity  :
    ?id_index:('ext index) ->
    config ->
    source ->
    dtd ->
    'ext spec ->
        'ext node
```

Parses a fragment, and validates the element.

```
val parse_wfcontent_entity :
    config ->
    source ->
    'ext spec ->
        'ext node
```

Parses a fragment, but checks it only on well-formedness.

4.4.3. Configuration options

```
type config =
    { warner : collect_warnings;
      errors_with_line_numbers : bool;
      enable_pinstr_nodes : bool;
      enable_super_root_node : bool;
      enable_comment_nodes : bool;
      encoding : rep_encoding;
      recognize_standalone_declaration : bool;
```

```
    store_element_positions : bool;
    idref_pass : bool;
    validate_by_dfa : bool;
    accept_only_deterministic_models : bool;
    ...
  }
```

- `warner`:The parser prints warnings by invoking the method `warn` for this warner object. (Default: all warnings are dropped)
- `errors_with_line_numbers`:If true, errors contain line numbers; if false, errors contain only byte positions. The latter mode is faster. (Default: true)
- `enable_pinstr_nodes`:If true, the parser creates extra nodes for processing instructions. If false, processing instructions are simply added to the element or document surrounding the instructions. (Default: false)
- `enable_super_root_node`:If true, the parser creates an extra node which is the parent of the root of the document tree. This node is called super root; it is an element with type `T_super_root`. - If there are processing instructions outside the root element and outside the DTD, they are added to the super root instead of the document. - If false, the super root node is not created. (Default: false)
- `enable_comment_nodes`:If true, the parser creates nodes for comments with type `T_comment`; if false, such nodes are not created. (Default: false)
- `encoding`:Specifies the internal encoding of the parser. Most strings are then represented according to this encoding; however there are some exceptions (especially `ext_id` values which are always UTF-8 encoded). (Default: ‘Enc_iso88591)
- `recognize_standalone_declaration`: If true and if the parser is validating, the `standalone="yes"` declaration forces that it is checked whether the document is a standalone document. - If false, or if the parser is in well-formedness mode, such declarations are ignored. (Default: true)
- `store_element_positions`: If true, for every non-data node the source position is stored. If false, the position information is lost. If available, you can get the positions of nodes by invoking the `position` method. (Default: true)
- `idref_pass`:If true and if there is an ID index, the parser checks whether every IDREF or IDREFS attribute refer to an existing node; this requires that the parser traverses the whole doument tree. If false, this check is left out. (Default: false)
- `validate_by_dfa`:If true and if the content model for an element type is deterministic, a deterministic finite automaton is used to validate whether the element contents match the content model of the type. If false, or if a DFA is not available, a backtracking algorithm is used for validation. (Default: true)
- `accept_only_deterministic_models`: If true, only deterministic content models are accepted; if false, any syntactically correct content models can be processed. (Default: true)

4.4.4. Which configuration should I use?

First, I recommend to vary the default configuration instead of creating a new configuration record. For instance, to set `idref_pass` to `true`, change the default as in:

```
let config = { default_config with idref_pass = true }
```

The background is that I can add more options to the record in future versions of the parser without breaking your programs.

Do I need extra nodes for processing instructions? By default, such nodes are not created. This does not mean that the processing instructions are lost; however, you cannot find out the exact location where they occur. For example, the following XML text

```
<x><?pi1?><y/><?pi2?></x>
```

will normally create one element node for `x` containing *one* subnode for `y`. The processing instructions are attached to `x` in a separate hash table; you can access them using `x # pinstr "pi1"` and `x # pinstr "pi2"`, respectively. The information is lost where the instructions occur within `x`.

If the option `enable_pinstr_nodes` is turned on, the parser creates extra nodes `pi1` and `pi2` such that the subnodes of `x` are now:

```
x # sub_nodes = [ pi1; y; pi2 ]
```

The extra nodes contain the processing instructions in the usual way, i.e. you can access them using `pi1 # pinstr "pi1"` and `pi2 # pinstr "pi2"`, respectively.

Note that you will need an exemplar for the PI nodes (see `make_spec_from_alist`).

Do I need a super root node? By default, there is no super root node. The `document` object refers directly to the node representing the root element of the document, i.e.

```
doc # root = r
```

if `r` is the root node. This is sometimes inconvenient: (1) Some algorithms become simpler if every node has a parent, even the root node. (2) Some standards such as XPath call the "root node" the node whose child represents the root of the document. (3) The super root node can serve as a container for processing instructions outside the root element. Because of these reasons, it is possible to create an extra super root node, whose child is the root node:

```
doc # root = sr          &&
sr # sub_nodes = [ r ]
```

When extra nodes are also created for processing instructions, these nodes can be added to the super root node if they occur outside the root element (reason (3)), and the order reflects the order in the source text.

Note that you will need an exemplar for the super root node (see `make_spec_from_alist`).

What is the effect of the UTF-8 encoding? By default, the parser represents strings (with few exceptions) as ISO-8859-1 strings. These are well-known, and there are tools and fonts for this encoding.

However, internationalization may require that you switch over to UTF-8 encoding. In most environments, the immediate effect will be that you cannot read strings with character codes >= 160 any longer; your terminal will only show funny glyph combinations. It is strongly recommended to install Unicode fonts (GNU Unifont (http://czyborra.com/unifont/), Markus Kuhn's fonts (http://www.cl.cam.ac.uk/~mgk25/download/ucs-fonts.tar.gz)) and terminal emulators that can handle UTF-8 byte sequences (http://myweb.clark.net/pub/dickey/xterm/xterm.html). Furthermore, a Unicode editor may be helpful (such as Yudit (ftp://metalab.unc.edu/pub/Linux/apps/editors/X/)). There are also FAQ (http://www.cl.cam.ac.uk/~mgk25/unicode.html) by Markus Kuhn.

By setting `encoding` to `` `Enc_utf8 `` all strings originating from the parsed XML document are represented as UTF-8 strings. This includes not only character data and attribute values but also element names, attribute names and so on, as it is possible to use any Unicode letter to form such names. Strictly speaking, PXP is only XML-compliant if the UTF-8 mode is used; otherwise it will have difficulties when validating documents containing non-ISO-8859-1-names.

This mode does not have any impact on the external representation of documents. The character set assumed when reading a document is set in the XML declaration, and character set when writing a document must be passed to the `write` method.

How do I check that nodes exist which are referred by IDREF attributes? First, you must create an index of all occurring ID attributes:

```
let index = new hash_index
```

This index must be passed to the parsing function:

```
parse_document_entity
  ~id_index:(index :> index)
  config source spec
```

Next, you must turn on the `idref_pass` mode:

```
let config = { default_config with idref_pass = true }
```

Note that now the whole document tree will be traversed, and every node will be checked for IDREF and IDREFS attributes. If the tree is big, this may take some time.

What are deterministic content models? These type of models can speed up the validation checks; furthermore they ensure SGML-compatibility. In particular, a content model is deterministic if the parser can determine the actually used alternative by inspecting only the current token. For example, this element has non-deterministic contents:

```
<!ELEMENT x ((u,v) | (u,y+) | v)>
```

If the first element in `x` is `u`, the parser does not know which of the alternatives `(u,v)` or `(u,y+)` will work; the parser must also inspect the second element to be able to distinguish between the alternatives. Because such look-ahead (or "guessing") is required, this example is non-deterministic.

The XML standard demands that content models must be deterministic. So it is recommended to turn the option `accept_only_deterministic_models` on; however, PXP can also process non-deterministic models using a backtracking algorithm.

Deterministic models ensure that validation can be performed in linear time. In order to get the maximum benefits, PXP also implements a special validator that profits from deterministic models; this is the deterministic finite automaton (DFA). This validator is enabled per element type if the element type has a deterministic model and if the option `validate_by_dfa` is turned on.

In general, I expect that the DFA method is faster than the backtracking method; especially in the worst case the DFA takes only linear time. However, if the content model has only few alternatives and the alternatives do not nest, the backtracking algorithm may be better.

4.5. Updates

Some (often later added) features that are otherwise not explained in the manual but worth to be mentioned.

- Methods node_position, node_path, nth_node, previous_node, next_node for nodes: See pxp_document.mli
- Functions to determine the document order of nodes: compare, create_ord_index, ord_number, ord_compare: See pxp_document.mli