NAME

flexml – generate validating XML processor and applications from DTD

SYNOPSIS

flexml [**–ASHDvdnLXV**] [**–s***skel*] [**–p***pubid*] [**–u***uri*] [**–r***rootags*] [**–a***actions*] *name*[*.dtd*]

DESCRIPTION

Flexml reads *name.dtd* which must be a DTD (Document Type Definition) describing the format of XML (Extensible Markup Language) documents, and produces a ‘‘validating'' XML *processor* with an interface to support XML *application*s. Proper applications can be generated optionally from special ‘‘action files'', either for linking or textual combination with the processor.

The generated processor will only validate documents that conform strictly to the DTD, *without extending it*, more precisely we in practice restrict XML rule [28] to

```
[28r] doctypedecl ::= '<!DOCTYPE' S Name S ExternalID S? '>'
```

where the `ExternalId` denotes the used DTD. (One might say, in fact, that *flexml* implements ‘‘non-extensible'' markup. :)

The generated processor is a *flex*(1) scanner, by default named *name.l* with a corresponding C header file *name.h* for separate compilation of generated applications. Optionally *flexml* takes an *actions* file with per-element actions and produces a C file with element functions for an XML application with entry points called from the XML processor (it can also fold the XML application into the XML processor to make stand-alone XML applications but this prevents sharing of the processor between applications).

In the `OPTIONS` entry elsewhere in this document we list the possible options, in the section on */ACTION FILE FORMAT* we explain how to write applications, in the `COMPILATION` entry elsewhere in this documentwe explain how to compile produced processors and applications into executables, and in the `BUGS` entry elsewhere in this documentwe list the current limitations of the system before giving standard references.

OPTIONS

Flexml takes the following options.

–A Generate a *stand-alone* scanner application. If combined with **–a***actions* then the application will be named as *actions* with the extension replaced by *.l*, otherwise it will be in *name.l*. Conflicts with **–S**, **–H**, and **–D**.

-a *actions*
Uses the *actions* file to produce an XML application in the file with the same name as *actions* after replacing the extension with *.c*. If combined with **–A** then instead the stand-alone application will include the action functions.

-D Generate a dummy application *name–dummy.c* with just empty functions to be called by the XML processor. If combined with **–a***actions* then the application will insert the specified actions and be named as *actions* with the extension replaced by *.c*. Conflicts with **–A**; implied by **–a** unless either of **–SHD** is specified.

-d Turns on debug mode in the flex scanner and also prints out the details of the DTD analysis performed by *flexml*.

-H Generate the header file *name.h*. Conflicts with **–A**; on by default if none of **–SHD** specified.

-L Makes the XML processor (as produced by *flex*(1)) count the lines in the input and keep it available to XML application actions in the integer `yylineno`. (This is off by default as the performance overhead is significant.)

-n ‘‘Dry-run": do not produce any of the output files.

-p *pubid*
Sets the document type to be `PUBLIC` with the identifier *pubid* instead of `SYSTEM`, the default.

-r *roottags*
Restricts the XML processor to validate only documents with one of the root elements listed in the comma-separated *roottags*.

-S Generate the scanner *name.l*. Conflicts with **–A**; on by default if none of **–SHD** specified.

-s *skel*
Use the skeleton scanner *skel* instead of the default.

-u *uri*
Sets the URI of the DTD, used in the `DOCTYPE` header, to the specified *uri* (the default is the DTD name).

-v Be verbose: echo each DTD declaration (after parameter expansion).

-V Print the version of *flexml* and exit.

ACTION FILE FORMAT

Action files, passed to the **–a** option, are XML documents conforming to the DTD *flexml-act.dtd* which is the following:

```
<!ELEMENT actions ((top|start|end)*,main?)>
<!ENTITY % C-code "(#PCDATA)">
<!ELEMENT top   %C-code;>
<!ELEMENT start %C-code;>  <!ATTLIST start tag NMTOKEN #REQUIRED>
<!ELEMENT end   %C-code;>  <!ATTLIST end   tag NMTOKEN #REQUIRED>
<!ELEMENT main  %C-code;>
```

The elements should be used as follows:

`top`
Use for top-level C code such as global declarations, utility functions, etc.

`start`
Attaches the code as an action to the element with the name of the required "`tag`" attribute. The "`%C-code;`" component should be C code suitable for inclusion in a C block (i.e., within `{`...`}` so it may contain local variables); furthermore the following extensions are available:

`{`*attribute*`}`: Can be used to access the value of the *attribute* as set with *attribute*=*value* in the start tag. In C, `{`*attribute*`}` will be interpreted depending on the declaration of the attribute. If the attribute is declared as an enumerated type like

```
<!ATTLIST attrib (alt1 | alt2 |...) ...>
```

then the C attribute value is of an enumerated type with the elements written `{`*attribute*=*alt1*`}`, `{`*attribute*=*alt2*`}`, etc.; furthermore an *unset* attribute has the "value" `{!`*attribute*`}`. If the attribute is not an enumeration then `{`*attribute*`}` is a null-terminated C string (of type `char*`) and `{!`*attribute*`}` is `NULL`.

`end`
Similarly attaches the code as an action to the end tag with the name of the required "`tag`" attribute; also here the "`%C-code;`" component should be C code suitable for inclusion in a C block. In case the element has "Mixed" contents, i.e, was declared to permit `#PCDATA`, then the following variable is available:

`{#PCDATA}`: Contains the text (`#PCDATA`) of the element as a null-terminated C string (of type `char*`). In case the Mixed contents element actually mixed text and child elements then `pcdata` contains the plain concatenation of the text fragments as one string.

`main`
Finally, an optional "`main`" element can contain the C `main` function of the XML application. Normally the `main` function should include (at least) one call of the XML processor:

`yylex()`: Invokes the XML processor produced by *flex*(1) on the XML document found on the standard input (actually the `yyin` file handle: see the manual for *flex*(1) for information on how to change this as well as the name `yylex`).

If no `main` action is provided then the following is used:

```
int main() { exit(yylex()); }
```

It is advisable to use XML `<![CDATA[` ... `]]>` sections for the C code to make sure that all characters are properly passed to the output file.

Finally note that *Flexml* handles empty elements <*tag*/> as equivalent to <*tag*></*tag*>.

COMPILATION

The following *make*(1) file fragment shows how one can compile *flexml*–generated programs:

```
# Programs.
FLEXML = flexml -v

# Generate linkable XML processor with header for application.
%.l %.h: %.dtd
        $(FLEXML) $<

# Generate C source from flex scanner.
%.c:    %.l
        $(FLEX) -Bs -o"$@" "$<"

# Generate XML application C source to link with processor.
# Note: The dependency must be of the form "appl.c: appl.act proc.dtd".
%.c:    %.act
        $(FLEXML) -D -a $^

# Direct generation of stand-alone XML processor+application.
# Note: The dependency must be of the form "appl.l: appl.act proc.dtd".
%.l:    %.act
        $(FLEXML) -A -a $^
```

BUGS

The present version of *flexml* is to be considered in ''early beta'' state thus bugs should be expected (and the author would like to hear about them). Here are some known restrictions that we hope to overcome in the future:

- The character set is merely ASCII (actually *flex*(1) handles 8 bit characters but only the ASCII character set is common with the XML default UTF−8 encoding).
- `ID` type attributes are not validated for uniqueness; `IDREF` and `IDREFS` attributes are not validated for existence.
- The `ENTITY` and `ENTITIES` attribute types are not supported.
- `NOTATION` declarations are not supported.
- The various `xml:`−attributes are treated like any other attributes; in particular `xml:spaces` should be supported.
- The XML processor currently uses a fixed-size buffer to read `pcdata`. It should not.
- The DTD parser is presently a perl hack so it may parse some DTDs badly; in particular the expansion of parameter entities may not conform fully to the XML specification.

- A child should be able to ''return'' a value for the parent (also called a *synthesised attribute*). Similarly an element in Mixed contents should be able to inject text into the `pcdata` of the parent.

FILES

/usr/share/flexml/skel
The skeleton scanner with the generic parts of XML scanning.

/usr/share/doc/flexml/
License, further documentation, and examples.

SEE ALSO

flex(1), Extensible Markup Language (XML) 1.0 (W3C Recommendation REC−xml-1998-0210).

AUTHOR

Flexml was written by Kristoffer Høgsbro Rose, `<krisrose@debian.org>`.

COPYRIGHT

The program is Copyright (c) 1999 Kristoffer Rose (all rights reserved) and distributed under the GNU General Public License (GPL, also known as ''copyleft'', which clarifies that the author provides absolutely no warranty for *flexml* and ensures that *flexml* is and will remain available for all uses, even comercial).

ACKNOWLEDGEMENT

I am grateful to NTSys (France) for supporting the development of *flexml*. Finally extend my severe thanks to Jef Poskanzer, Vern Paxson, and the rest of the *flex* maintainers and GNU developers for a great tool.