Kodowanie, przetwarzanie i formatowanie dokumentów SGML-owych

Piotr Bolek
ul. Szkolna 15
05-180 Pomiechówek
Phone: (48) 22 – 785 43 39
P.Bolek@GUST.org.pl

Streszczenie

Tekst ten jest próbą opisania doświadczeń związanych z praktycznym wykorzystaniem SGML-a do kodowania dokumentów. Nie ma tu gotowego przepisu na formatowanie dowolnych dokumentów SGML-owych w dowolny sposób, ale opis różnego rodzaju doświadczeń zebranych przy praktycznym wykorzystaniu SGML-a w ciagu kilku lat pracy z różnymi dokumentami. Opisane są przydatne w praktyce konstrukcje SGML-owe (przetwarzanie warunkowe, definicje, skróty itp.). Przedstawiony jest także zestaw dostępnych nieodpłatnie narzędzi, które są z powodzeniem wykorzystywane przez autora do formatowania dokumentów kodowanych w SGML-u. W tekście nie ma informacji, które można znaleźć w dokumentacji do opisywanego oprogramowania, np. standardowych przepisów na poprawne zainstalowanie pakietów.

Wprowadzenie

Standard SGML na dobre już zadomowił się we współczesnym świecie. Doceniono jego zalety i możliwości. Coraz częściej spotyka się oprogramowanie, którego dokumentacja jest tworzona w SGML-u i udostępniana w różnych formatach prezentacyjnych. Podstawowym problemem, na jaki mogą napotkać potencjalni lub faktyczni użytkownicy SGML-a, jest jednak może nie brak, ale trudność w dotarciu do gotowych, łatwych do użycia narzędzi, takich jak szablony dla różnych typów dokumentów czy systemy formatujące. Większość autorów piszących o SGML-u ogranicza się do opisania formalnej struktury dokumentu SGML-owego lub sposobu konstruowania Definicji Typu Dokumentu. W praktyce przeciętny użytkownik zwykle sam nie tworzy DTD więc nie bardzo interesują go konstrukcje do tego używane, za to chciałby mieć odpowiadającą jego potrzebom gotową definicję. Z drugiej strony wszyscy użytkownicy zetknęli się z dokumentami HTML-owymi, wiedzą więc, na czym polega sposób znakowania dokumentów SGML-owych. Poza tym, elementarna wiedza na temat konstrukcji używanych przy tworzeniu DTD przydaje się w praktyce, jeśli np. potrzebujemy dodać do gotowego DTD jeden element, albo zdefiniować sobie wygodne skróty.

Do efektywnej pracy autorzy dokumentów potrzebują odpowiedniego zestawu narzędzi. Pierwszym elementem takiego zestawu jest edytor strukturalny wspomagający oznaczanie struktury tekstu. Kolejnym niezbędnym do pracy z SGML-em narzędziem jest parser SGML-owy. Służy on do weryfikacji poprawności dokumentu i może być wbudowany w edytor, często jest to jednak zewnętrzny program, który może być także używany w procesie przetwarzania bądź formatowania dokumentu. Ostatnim elementem warsztatu są narzędzia do przetwarzania i formatowania dokumentów: różnego rodzaju szablony i programy do transformowania zawartości dokumentu.

Przydatne w praktyce konstrukcje SGML-owe

Zaczniemy jednak nie od narzędzi, a od formatu. Chcąc efektywnie wykorzystać SGML warto wiedzieć o nim co nieco. Zakładam, że podstawową wiedzę na temat SGML-a ma każdy w miarę świadomy użytkownik komputera. Tutaj chciałbym pokazać tylko kilka ciekawych i moim zdaniem przydatnych konstrukcji, jakich można używać pracując z SGML-em.

Rozszerzanie i dostosowywanie DTD. Większość konstrukcji, o których będzie mowa poniżej, jest przeznaczona do użycia nie bezpośrednio w konkretnym dokumencie, ale w definicji typu dokumentu. Wspomniałem już jednak, że przeciętny użytkownik nie będzie zwykle sam tworzył DTD. Dlatego potrzebny jest mechanizm umożliwiający modyfikację, dostosowanie lub drobne rozszerzenia DTD bez konieczności ingerowania w zawartość pliku z definicją. Oczywiście, mechanizm taki został przewidziany w standardzie SGML. Lokalne rozszerzenia DTD, definicje skrótów, zapisanych w zewnętrznych plikach fragmentów dokumentu, można umieszczać w nagłówku dokumentu w tzw. podzbiorze deklaracji typu dokumentu (ang. *document type declaration subset*):

```
<!DOCTYPE ARTICLE PUBLIC
"-//Davenport//DTD DocBook V3.0//EN"
  [ -- To jest podzbiór deklaracji
       typu dokumentu -- ]>
```

Konkretne przykłady tego, co można wpisać w tym miejscu, będą się pojawiały w dalszej części tekstu.

Definicje i nie tylko (*entity*). Podstawową konstrukcją, przydającą się w każdym dokumencie są definicje, które w SGML-u nazywają się encjami (ang. *entity*) i zapewniają większą funkcjonalność niż proste definiowanie skrótów. Pozwalają one definiować pewne stałe „jednostki tekstowe" (bez możliwości parametryzacji w sensie wywoływania z różnymi argumentami, jak jest to np. w przypadku makrodefinicji TEX-owych). Niemniej, możliwości jakie dają encje są dosyć ciekawe i pożyteczne. W standardzie SGML są zdefiniowane trzy rodzaje encji: ogólne (ang. *general*) i parametryzujące (ang. *parameter*) oraz odwołania znakowe (ang. *character references*). Encje pierwszego typu umożliwiają tworzenie definicji. Umożliwiają także definiowanie i włączanie zapisanych w oddzielnych plikach części dokumentu.

```
<!DOCTYPE ARTICLE PUBLIC
"-//Davenport//DTD DocBook V3.0//EN" [
<!ENTITY GUST "Polska Grupa Użytkowników
                Systemu &TeX;">
<!ENTITY TeX SDATA "\TeX{}">
<!ENTITY example system "ex1.sgml">
]>
```

W powyższym przykładzie zdefiniowane są trzy różne ogólne encje. Pierwsza jest definicją skrótu w pliku źródłowym – zamiast pisać pełną nazwę, możemy używać skrótu (`&GUST;`). Druga, do której odwołanie znajduje się w pierwszej encji, definiuje logo TEX. Encja ta jest zdefiniowana jako `SDATA`, co oznacza, że jej rozwinięcie jest specyficzne dla systemu przetwarzającego (w tym przypadku TEX-a). Dla innego systemu przetwarzającego należy zdefiniować ją inaczej.

Trzecia encja to odwołanie do zewnętrznego pliku zawierającego fragment dokumentu. Użycie jej w tekście dokumentu spowoduje wczytanie zawartości pliku `ex1.sgml` (mechanizm funkcjonalnie identyczny jak `#include` w języku C czy `\input` w TEX-u).

Odwołania znakowe mogą być używane do wstawienia w tekście dokumentu znaków, które normalnie są rozpoznawane jako znaczniki (np. „`<`" albo „`&`"). W tym celu w dokumencie wpisujemy następujące odwołanie „`&#`⟨*num*⟩`;`", gdzie ⟨*num*⟩ jest dziesiętnym kodem znaku. Nasze przykładowe znaki zostały więc zapisane następująco: „`<`" jako „`&#60`", a „`&`" jako „`&#38`".

Encje parametryzujące służą, jak sama nazwa wskazuje, do parametryzacji dokumentu. Nie mogą być one używane bezpośrednio w tekście, a tylko w części „znacznikowej" dokumentu (np. do oznaczenia fragmentów tekstu, które mają być przetwarzane warunkowo). Mogą być także używane do zdefiniowania odwołań do zapisanych w zewnętrznych plikach standardowych lub niestandardowych zestawów encji, fragmentów DTD lub lokalnych definicji. Mogą także służyć do parametryzacji DTD. Na przykład bardzo sensownie sparametryzowane jest DTD DocBook. Można tam, bez konieczności modyfikowania oryginalnego pliku, dodawać np. własne elementy.

```
<!DOCTYPE ARTICLE PUBLIC
"-//Davenport//DTD DocBook V3.0//EN" [
<!ENTITY % my.ext SYSTEM "komp-pb.ext">
%my.ext

<!ENTITY % local.para.class
           "|TestQuestion|Solution">
]>
```

W powyższym przykładzie zdefiniowano dwie encje. Pierwsza z nich `my.ext` służy do dołączenia do dokumentu definicji zawartych w pliku `komp-pb.ext`. Druga jest wykorzystaniem parametrycznej encji `local.para.class`, umożliwiającej dodanie do struktury dokumentu własnych elementów.

Różne rodzaje danych tekstowych. Podstawowymi danymi w dokumentach, w tym oczywiście w dokumentach SGML-owych, są dane tekstowe. W SGML-u istnieją jednak dane tekstowe różnego typu, których znaczenie, a raczej sposób traktowania przez parser i system formatujący mogą być różne. Większość tekstu w dokumencie to `PCDATA` (ang. *parsed character data*), czyli zwykłe dane tekstowe, w których mogą się pojawić i są interpretowane znaczniki elementów oraz odwołania do encji ogólnych. Inaczej jest traktowany tekst oznaczony jako `CDATA`. Tutaj nie są rozwijane żadne definicje (encje), nie są też rozpoznawane żadne znaczniki. Dzięki temu możliwe jest dosłowne wpisywanie tekstu, który inaczej byłby potraktowany jako znaczniki. Jeszcze innym rodzajem tekstu jest `RCDATA`. W tekście oznaczonym jak `RCDATA` nie są rozpoznawane znaczniki SGML-owe, natomiast są rozwijane encje oraz odwołania znakowe (ang. *character reference*).

Poniższy przykład pokazuje różnice między tekstem typu CDATA i RCDATA

```
<programlisting>
<![CDATA[Tutaj <emph>znaczniki</emph>
i &encje; nie są rozwijane]]>

<![RCDATA[Tutaj <emph>znaczniki</emph>
nie są interpretowane, ale
&encje; są rozwijane]]>
</programlisting>
```

Zakładając, że encja encje była zdefiniowana jako: <!ENTITY encje "ENCJE">, to po przetworzeniu i powyższego przykładu moglibyśmy otrzymać następujący efekt:

```
Tutaj <emph>znaczniki</emph>
i &encje; nie są rozwijane

Tutaj <emph>znaczniki</emph>
nie są interpretowane, ale
ENCJE są rozwijane
```

Wreszcie wspomniany już powyżej tekst typu SDATA, używany jest najczęściej dla zdefiniowania znaków, lub sekwencji, których sposób interpretacji (lub notacji) jest zależny od systemu. Większość symboli matematycznych i znaków drukarskich została zdefiniowana jako encje typu SDATA i są dostępne w standardowych, zewnętrznych zbiorach encji (używanych domyślnie np. w DTD DocBook). Przykładowo do znaków „<” i „&” można odwołać się w dokumencie w standardowy sposób: „<” oraz „&”. Sposób ten jest nawet bardziej przenośny niż użyte wcześniej odwołania znakowe, ponieważ nie zakłada konkretnego kodowania znaków.

```
<!ENTITY lt  SDATA "[lt    ]"
           --=less-than sign R:-->
<!ENTITY amp SDATA "[amp   ]"
           --=ampersand-->
```

Przetwarzanie warunkowe i nie tylko (*marked sections*). Czasami zachodzi potrzeba sparametryzowania dokumentu. Jeśli chcemy lub musimy uwzględnić specyficzne cechy systemu, którego będziemy używali do formatowania, albo prezentacji dokumentu, możemy użyć konstrukcji zwanej oznaczoną sekcją (ang. *marked section*). Konstrukcji tej używaliśmy już wcześniej, oznaczając pewne fragmenty dokumentu jako CDATA albo RCDATA. Sekcje oznaczone definiuje się następująco:

<![⟨*rodzaj*⟩[⟨*zawartość*⟩]]>, gdzie ⟨*rodzaj*⟩ to CDATA, RCDATA, INCLUDE, IGNORE albo odwołanie do parametryzującej encji, rozwijającej się do jednego z tych słów.

Jak już wiemy, sekcje oznakowane jako CDATA i RCDATA są interpretowane inaczej niż reszta tekstu. Sekcje oznaczone jako IGNORE są pomijane w przetwarzaniu, a oznaczone jako INCLUDE „przechodzą” bez zmian. Sekcje, w których użyjemy zdefiniowanych encji parametryzujących, umożliwiają warunkowe przetwarzanie dokumentu. Popatrzmy na poniższy fragment deklaracji:

```
<!ENTITY % HTML "IGNORE">
<![%HTML[
<!ENTITY % TEX "IGNORE">
]]>
<!ENTITY % TEX "INCLUDE">
<![%TEX[
<!ENTITY TeX SDATA "\TeX{}">
]]>
<![%HTML[
<!ENTITY TeX SDATA "TeX">
]]>
```

W przykładzie tym są zdefiniowane dwie encje parametryzujące: %HTML i %TEX oraz jedna encja ogólna TeX. Encje parametryzujące zostały tutaj użyte zgodnie ze swoją nazwą. Domyślnie – %TEX rozwija się do INCLUDE, a %HTML do IGNORE. Czyli nasz przykład można by zapisać tak:

```
<![IGNORE[
<!ENTITY % TEX "IGNORE">
]]>
<![INCLUDE[
<!ENTITY TeX SDATA "\TeX{}">
]]>
<![IGNORE[
<!ENTITY TeX SDATA "TeX">
]]>
```

Ostatecznie parser potraktuje to tak, jakby na wejściu pojawiło się:

```
<!ENTITY TeX SDATA "\TeX{}">
```

Możemy jednak wywołując parser (mowa o parserze nsgmls) podać mu opcję -i HTML. Oznacza to, że w momencie analizowania dokumentu encja parametryzująca %HTML będzie zawczasu zdefiniowana jako INCLUDE. Ponieważ w SGML-u wcześniejsza definicja encji „przysłania” późniejszą, to włączona zostanie pierwsza sekcja oznaczona jako %HTML, w której encja %TEX jest zdefiniowana jako IGNORE. Kolejna definicja %TEX (jako INCLUDE) zostanie pominięta i w efekcie nasz przykład zostanie zinterpretowany jakbyśmy napisali:

```
<!ENTITY TeX SDATA "TeX">
```

Użyliśmy tu wzajemnie się wykluczających encji parametryzujących %TEX oraz %HTML, które służą w naszym przykładzie do wyboru odpowiedniej dla sposobu prezentacji definicji encji &TeX.

Skróty. Innym przydatnym mechanizmem są skróty (ang. *short references*). Mechanizm skrótów pozwala definiować w pewnych kontekstach specjalne konwencje notacyjne. Dla użytkowników TEX-a np. wygodnie byłoby oznaczać myślnik przez wpisanie dwóch znaków minus „`--`” zamiast odwołania do ogólnej encji `&mdash` albo zdefiniowanie znaku tyldy „`~`” jako odwołania do niełamliwego odstępu (` `). Używając mechanizmu skrótów można też zdefiniować np. znaki cudzysłowów (`"`) w ten sposób, że pierwszy napotkany cudzysłów powoduje otwarcie elementu `quote` (cytat), a drugi zamknięcie tego elementu.

Definicja skrótu musi być umieszczona w DTD, albo w podzbiorze deklaracji typu dokumentu. Oto przykład definicji skrótów wymienionych powyżej.

```
<!DOCTYPE ARTICLE PUBLIC
"-//Davenport//DTD DocBook V3.0//EN" [
<!ENTITY q-start "<QUOTE>">
<!ENTITY q-end  "</QUOTE>">
<!SHORTREF PARA-MAP
               "--" mdash
               "~"  nbsp
               '"'  q-start >
<!SHORTREF QUOTE-MAP
               "--" mdash
               "~"  nbsp
               '"'  q-end >
<!USEMAP PARA-MAP  PARA>
<!USEMAP QUOTE-MAP QUOTE>
]>
```

Definicja skrótu składa się jak widać z trzech części:

1. Definicja encji ogólnej, której wartość będzie użyta jako tekst zastępujący dany skrót. W naszym przykładzie zdefiniowaliśmy dwie encje, które w zależności od kontekstu zastępują znak cudzysłowu. Encje `&mdash` i ` `, używane do zastąpienia skrótów „`--`” i „`"`” są zdefiniowane domyślnie we wczytywanych w DTD zbiorach encji, nie musimy więc ich podawać tutaj jawnie.
2. Definicja mapowania skrótów (ang. *short reference map*). W naszym przykładzie zdefiniowaliśmy dwie mapy, które będziemy używać w zależności od kontekstu.
3. Określenie kontekstu, w którym ma być używana mapa (ang. *short reference use*). U nas domyślnie jest używana mapa `PARA-MAP`. Po napotkaniu skrótu „`"`” (lub jawnie podanej etykiety otwierającej elementu `quote`) następuje przełączenie na mapę `QUOTE-MAP`, w której skrót „`"`” jest zdefiniowany jako etykieta zamykająca element `quote`.

Przy tak zdefiniowanych skrótach następujący fragment tekstu:

```
<para>"Cytat" i~twarda
spacja -- przykład użycia
skrótów.</para>
```

jest w dokładnie równoważny takiemu:

```
<PARA><QUOTE>Cytat</QUOTE> i twarda
spacja &mdash; przykład użycia
skrótów.</PARA>
```

Co nieco o DSSSL

DSSSL (ang. *Document Style Semantics and Specification Language*) jest, jak stwierdził Sebastian Rahtz, powodem frustracji wśród ludzi ze środowisk, w których wykorzystuje się SGML. Z jednej strony jest to długo oczekiwany efekt wieloletniej pracy, która zakończyła się opracowaniem bardzo ciekawego i ogólnego modelu przetwarzania i formatowania tekstu. Z drugiej jednak strony złożoność tego modelu ma kilka poważnych konsekwencji:

- nie istnieje (i nie wiadomo kiedy i czy w ogóle powstanie) pełna implementacja standardu;
- nie istnieją (i nie wiadomo kiedy i czy w ogóle powstaną) systemy formatujące, spełniające wszystkie wymagania tego standardu;
- podjęto decyzję, że do do formatowania dokumentów XML-owych będzie wykorzystany prostszy język stylów – XSL.

Problemy z DSSSL-em są także spowodowane dosyć egzotycznym językiem, w którym trzeba pisać specyfikacje przetwarzania i formatowania. Język ten jest zdefiniowany w oparciu o język funkcyjny Scheme. Dla zwykłych śmiertelników może to być poważną przeszkodą.

Wspomniane powyżej trzy fakty mogą znacznie ograniczyć użyteczność i praktyczne znaczenie tego standardu. Pomimo przeciwności wydaje się, że DSSSL może być z powodzeniem wykorzystywany w ograniczonym zakresie w praktyce. Jest to zasługą głównie dwóch osób: Jamesa Clarka, który zaimplementował duży podzbiór języka stylów DSSSL w postaci programu `jade` oraz Normana Walsha – autora DSSSL-owej specyfikacji formatowania dokumentów oznakowanych z wykorzystaniem DTD DocBook.

Potencjalne możliwości formatowania z wykorzystaniem specyfikacji DSSSL-owych są bardzo duże. Nawet obecnie, kiedy nie ma jeszcze kompletnych implementacji narzędzi DSSSL-owych, wykorzystanie tego, co już jest dostępne, może dawać wymierne korzyści. Zestaw DocBook DTD + Emacs/psgml + docbook-stylesheets + `jade` + Jadetex stanowi gotowe do użycia środowisko, w którym można stosunkowo prosto formatować dokumenty w różny sposób, bez konieczności pisania skomplikowanych systemów konwertujących. Co prawda jakość dokumentów drukowanych (zarówno za pośrednictwem Jadetex-a jak i ścieżki RTF-owej) stanowi słaby punkt takiego rozwiązania, ale są zastosowania, w których to nie przeszkadza.

Identyfikatory SGML-owe i katalogi

DTD, zbiory encji i szablony DSSSL-owe są przechowywane w plikach. Można się do nich jednak odwoływać w sposób niezależny od ich położenia w systemie i nazwy pliku, w którym są zapisane. W tym celu używa się tzw. publicznych identyfikatorów (ang. *formal public identifiers*). Identyfikatory te konstruuje się zgodnie z konwencją, której zasada nie jest istotna dla użytkowników gotowych DTD i zbiorów encji. Ważne jest aby programy przetwarzające dokumenty SGML-owe potrafiły przetłumaczyć identyfikatory na nazwy ścieżkowe w systemie plików. Mechanizm tłumaczenia identyfikatorów publicznych na nazwy plików jest taki sam w przypadku wszystkich narzędzi wspominanych w tym tekście (`nsgmls`, `jade` i Emacs). Wszystkie one korzystają z plików zwanych katalogami (ang. *catalog files*), zwyczajowo nazywających się `catalog` (taki plik umieszczony w bieżącym katalogu zostanie użyty automatycznie do mapowania), chociaż można je nazywać dowolnie. Pliki katalogowe zawierają wiersze postaci:

PUBLIC "⟨*identyfikator publiczny*⟩"
⟨*nazwa pliku*⟩

Nazwa pliku może być także podana w cudzysłowach. Poniżej są pokazane przykłady pozycji z plików katalogowych (przełamane aby zmieściły się w kolumnie – w rzeczywistości muszą pasować dokładnie do identyfikatorów używanych w dokumencie włącznie z liczbą spacji między wyrazami).

```
PUBLIC "-//Davenport//DTD DocBook
              V3.0//EN" "docbook.dtd"
PUBLIC "-//USA-DOD//DTD Table Model
            951010//EN" "cals-tbl.dtd"
PUBLIC "ISO 8879:1986//ENTITIES
         Publishing//EN" "iso-pub.ent"
PUBLIC "ISO 8879:1986//ENTITIES Greek
           Letters//EN" "iso-grk1.ent"
PUBLIC "-//Norman Walsh//DOCUMENT
         DocBook Print Stylesheet//EN"
         print/docbook.dsl
PUBLIC "-//Norman Walsh//DOCUMENT
          DocBook HTML Stylesheet//EN"
          html/docbook.dsl
```

Jak widać, w katalogach można podawać nazwy względne. Pozostaje więc pytanie, skąd programy mają wiedzieć gdzie szukać plików zawierających DTD oraz samych plików katalogowych. Mechanizm jest podobny jak w przypadku ścieżek do plików wykonywalnych – za pośrednictwem zmiennych środowiska. Używane są dwie zmienne `SGML_CATALOG_FILES` zawierająca listę ścieżek do plików katalogowych oraz `SGML_SEARCH_PATH` zawierająca listę katalogów, w których należy poszukiwać. Oto przykład ustawienia tych zmiennych w systemie Unix (shell Bourne'a):

```
SGML_SEARCH_PATH=$HOME/SGML/DOCBOOK:\
$HOME/SGML/ISO-8879:$HOME/SGML/HTML
SGML_CATALOG_FILES=$HOME/SGML/catalog
export SGML_SEARCH_PATH
export SGML_CATALOG_FILES
```

Podstawowe narzędzia

DocBook – definicja typu dokumentu. DTD nie jest narzędziem a jedynie określeniem języka jakiego będziemy używali do znakowania dokumentów. Jednak, ponieważ DTD nie piszemy sami lecz korzystamy z „gotowca", więc omówione będzie razem z narzędziami. DocBook jest bardzo obszernym DTD, przeznaczonym – jak sama nazwa wskazuje – do znakowania dokumentacji technicznej oraz książek, głównie z dziedziny informatyki. Jest przygotowane bardzo starannie i dobrze udokumentowane. Dużą zaletą jest modularność i możliwość łatwego rozszerzania tego DTD bez konieczności modyfikowania jego plików źródłowych. Twórcy zdefiniowali liczne parametryzujące encje pozwalające bardzo łatwo dołączać do standardowej struktury elementów nowe – własne.

DocBook nadaje się do zastosowania w małych dokumentach, takich jak artykuł do czasopisma, czy referat konferencyjny, jak i do publikacji dużych – np. licząca prawie tysiąc stron książka.

Dużą zaletą i zachętą do używania DocBooka jest dostępność gotowych DSSSL-owych szablonów stylów autorstwa Normana Walsha, umożliwiających generowanie z jednego źródła dokumentów drukowalnych (RTF, Jadetex, Frame Maker) oraz wersji HTML-owej bez konieczności pisania własnych narzędzi konwertujących.

Emacs jako edytor strukturalny. Komercyjne edytory strukturalne są stosunkowo drogie, ale na szczęście nie są jedynymi narzędziami, których można używać do pisania tekstów w sposób strukturalny. Autorzy wychowani na TEX-u mają dostępny zupełnie przyzwoity edytor strukturalny całkowicie za darmo. Jest to oczywiście nieoceniony Emacs pracujący w trybie PSGML. Pracując w tym trybie Emacs wczytuje definicję DTD, śledzi kontekst (czyli w jakim miejscu struktury dokumentu znajduje się autor) i podpowiada jakie są dopuszczalne znaczniki, atrybuty, lub inne elementy dokumentu SGML-owego dostępne w danym momencie. Dodatkowo wykorzystując tryb `font-lock` w różny sposób koloruje różne elementy struktury dokumentu.

Podstawowe możliwości jakie daje Emacs pracujący w trybie PSGML są następujące:

- pokazywanie miejsca, w którym znajduje się kursor w strukturze elementów (`C-c C-c`). Np. w *tym* miejscu w dokumencie Emacs wyświetlił:

```
#PCDATA in emphasis in para
  in listitem  in itemizedlist
  in sect2 in sect1 in article
```

- pokazywanie dopuszczalnych w danym miejscu elementów (`C-c C-t`). Np. każdy element listy (`<listitem>`), nie może zgodnie ze specyfikacją DocBook DTD zawierać bezpośrednio tekstu ale musi zawierać element niższego poziomu:

```
Current element: listitem
Element content: element
Current element can not end here
Valid start-tags
In current element: <abstract>
.. (45 elementów wycięto) ..
<warning>
```

  Dla porównania: wywołanie polecenia `C-c C-t` w elemencie `<para>` zawartym w `<listitem>` daje następujący efekt:

```
Current element: para
Element content: mixed
Current short reference map: paramap
Valid end-tags : </para>
Valid start-tags
In current element:  <abbrev>
.. (112 elementów usunięto) ..
<xref>
If omitting </para>:  <abstract>
.. (9 elementów usunięto)
  <simpara>
If omitting </listitem>:  <listitem>
```

- Kontekstowe wprowadzanie etykiet otwierających lub zamykających (`C-c <`), zamykanie otwartego elementu (`C-c /`), wstawianie całego elementu, czyli etykiety otwierającej i zamykającej (`C-c C-e`), oraz „otagowanie” regionu, czyli bloku tekstu (`C-c C-r`). Po podaniu każdego z tych poleceń Emacs podpowiada jakie są dopuszczalne w danym kontekście elementy (etykiety).
- Poruszanie się po strukturze dokumentu. Skakanie do następnego, poprzedniego, wyższego, niższego itd. elementu w strukturze. Konkretne skróty klawiszowe do sprawdzenia w dostępnej „on-line” pomocy.
- „Zwijanie” (ang. *folding*) i „rozwijanie” fragmentów struktury dokumentu. Emacs potrafi ukrywać całą zawartość dokumentu, zostawiając widoczny tylko pierwszy jego wiersz.

Te i liczne inne przydatne polecenia są dostępne także w systemie menu. Z rzeczy ciekawszych – możliwe jest zweryfikowanie poprawności dokumentu (tzn. zgodności z DTD). W tym celu Emacs wywołuje zewnętrzny parser SGML-owy (domyślnie jest to program `nsgmls` czyli parser SP Jamesa Clarka). Jeśli parser znajdzie błędy w strukturze dokumentu, to Emacs sam może odnaleźć miejsce, w którym został stwierdzony błąd.

SP – parser SGML-owy. Ważnym narzędziem wykorzystywanym przy pracy z dokumentami SGML-owymi jest parser. Wykorzystywany jest on nie tylko do weryfikacji poprawności dokumentów, ale także przy ich przetwarzaniu. W szczególności, jeśli zdecydujemy się na przetwarzanie dokumentu z wykorzystaniem biblioteki `SGMLSpm`, to konieczne będzie wykorzystanie parsera `nsgmls` wchodzącego w skład pakietu SP. Parser ten jest dostępny w postaci binarnej dla systemów MS oraz w źródłach dla systemów uniksowych. W pakiecie z parserem są dostępne także normalizatory SGML-owe, czyli programy, które umieszczają w dokumencie wszystkie opuszczone zgodnie z regułami minimalizacji znaczniki, rozwijają skróty itd. W zestawie jest też program `sx`, służący do konwersji dokumentu SGML-owego na XML.

Biblioteka `SGMLSpm` i program `sgmlspl`. Jeszcze przed ostatecznym ustaleniem standardu DSSSL bardzo dużą popularność uzyskał perlowy pakiet SGMLSpm oraz zbudowany z jego wykorzystaniem program `sgmlspl`. Biblioteka jest przeznaczona do tworzenia postprocesorów danych wychodzących z parsera SP, umożliwiając pisanie sterowanych zdarzeniami konwerterów dokumentów SGML-owych. Zdarzeniami pojawiającymi się na wejściu programu przetwarzającego są elementy struktury dokumentu SGML-owego. W bibliotece SGMLSpm zdefiniowanych jest kilka klas umożliwiających dostęp do różnych części dokumentu SGML-owego. Najważniejsze z nich to:

- `SGML_Event` – podstawowa klasa używana w czasie przetwarzania dokumentu. Obiekty klasy określają rodzaj zdarzenia, związane z nim dane, miejsce wystąpienia (plik, wiersz). Najważniejsze metody:
  - `type` – zwraca ciąg znaków określający typ zdarzenia (`start_element`, `end_element`, `cdata`, `sdata` itd.)
  - `data` – zwraca dane związane z danym zdarzeniem (typem danych może być ciąg znaków, obiekt klasy `SGMLS_Element` lub `SGMLS_Entity`)
  - `element` – zwraca element (obiekt klasy `SGMLS_Element`), określający kontekst zdarzenia
- `SGML_Element` – obiekty tej klasy są tworzone w chwili napotkania etykiety zamykającej lub otwie-

rającej. Elementy „znają” swoich „przodków” (elementy położone wyżej w hierarchii) i umożliwiają dostęp do wartości atrybutów;

- name – nazwa elementu (dużymi literami);
- parent – obiekt tej samej klasy odpowiadający elementowi o jeden poziom wyżej w hierarchii;
- attributes – zwraca tablicę asocjacyjną zawierającą atrybuty elementu; każdy element tablicy jest obiektem klasy SGMLS_Attribute;
- attribute_names – zwraca tablicę zawierającą nazwy atrybutów elementu;
- attribute(⟨*nazwa*⟩) – zwraca obiekt odpowiadający atrybutowi elementu o podanej nazwie;
- in(⟨*nazwa*⟩) – zwraca obiekt odpowiadający elementowi wyższego poziomu, jeśli jest on bezpośrednim przodkiem bieżącego elementu, lub pusty ciąg znaków, jeśli element o podanej nazwie nie jest bezpośrednim przodkiem bieżącego;
- within(⟨*nazwa*⟩) – zwraca obiekt odpowiadający elementowi wyższego poziomu, jeśli jest on przodkiem (być może pośrednim) bieżącego elementu, lub pusty ciąg znaków, jeśli element o podanej nazwie nie jest przodkiem bieżącego;

- SGMLS_Attribute – obiekty tej klasy umożliwiają dostęp do wartości atrybutów elementów SGML-owych;
  - name – nazwa atrybutu;
  - type – typ atrybutu;
  - value – wartość atrybutu (najczęściej ciąg znaków);
  - is_implied – zwraca wartość „prawda”, jeśli wartość atrybutu jest domyślna, lub „fałsz” jeśli była podana jawnie.

Biblioteki SGMLSpm można używać bezpośrednio, ale zwykle wygodniej jest skorzystać z programu sgmlspl. Program sgmlspl działa w oparciu o plik sterujący, który jest napisanym w perlu przepisem określającym jakie działania należy podjąć po napotkaniu różnych elementów dokumentu SGML-owego. W programie jest wykorzystywana biblioteka SGMLS::Output pozwalająca na wygodne zarządzanie potokiem danych generowanych przez program.

Program ten działa jako postprocesor dla parsera nsgmls i typowe wywołanie jest następujące:

```
nsgmls input.sgml | sgmlspl spec.pl \
                            > doc.latex
```

gdzie spec.pl jest plikiem zawierającym specyfikację przetwarzania dokumentu SGML-owego. Specyfikacja przetwarzania musi zawierać procedury obsługujące różne zdarzenia generowane przez konstrukcje SGML-owe zapisane w pliku źródłowym. Do definiowania procedury obsługi (ang. *handler*) służy dwuargumentowa funkcja sgml(⟨*zdarzenie*⟩, ⟨*działanie*⟩). Pierwszy argument określa zdarzenie – może to być zdarzenie *ogólne* – wtedy działanie określone przez drugi argument jest wykonywane dla każdego zdarzenia danego typu. Jeśli zdarzenie jest *specyficzne* (konkretne), to działanie jest wykonywane tylko dla konkretnego zdarzenia danego typu. Zdarzenia ogólne odpowiadają typom zwracanym przez metodę type obiektów klasy SGMLS_Event.

Zdarzenia specyficzne można określać następująco:

- '<ELEM>' – napotkanie etykiety otwierającej element o nazwie ELEM;
- '</ELEM>' – napotkanie etykiety zamykającej element o nazwie ELEM;
- '|TEKST|' – napotkanie tekstu typu SDATA.

Ciekawą cechą programu sgmlspl jest możliwość automatycznego utworzenia „szkieletu” pliku specyfikującego konwersję. Aby utworzyć taki szkielet należy wywołać program sgmlspl, podając jako specyfikację przetwarzania plik skel.pl rozprowadzany razem z pakietem.

Załóżmy, że mamy następujący prosty plik:

```
<!doctype chapter PUBLIC
"-//Davenport//DTD DocBook V3.0//EN">

<chapter id="example1">
<title>Przykładowy dokument</title>

<para>Początek rozdziału zawierającego
inne elementy.</para>
<itemizedlist>
<listitem>
<para>Pierwszy element listy
nienumerowanej</para>
</listitem>
<listitem>
<para>Drugi element</para>
</listitem>
</itemizedlist>
</chapter>
```

Po wykonaniu poniższego potoku:

```
nsgmls doc.sgml | sgmlspl skel.pl >  myskel.pl
```

otrzymamy szkielet specyfikacji przetwarzania, którego główna część jest pokazana poniżej:

```
# Document Handlers.
#
sgml('start', sub {});
sgml('end', sub {});

# Element Handlers.

# Element: CHAPTER
sgml('<CHAPTER>', "");
sgml('</CHAPTER>', "");

# Element: TITLE
```

```
sgml('<TITLE>', "");
sgml('</TITLE>', "");

# Element: PARA
sgml('<PARA>', "");
sgml('</PARA>', "");

# Element: ITEMIZEDLIST
sgml('<ITEMIZEDLIST>', "");
sgml('</ITEMIZEDLIST>', "");

# Element: LISTITEM
sgml('<LISTITEM>', "");
sgml('</LISTITEM>', "");
```

Jak widać dla każdego elementu w dokumencie wejściowym została wygenerowana para wywołań funkcji `sgml` pozwalająca na zdefiniowanie procedur obsługi dla występujących w dokumencie elementów. Domyślnym sposobem obsługi wszystkich zdarzeń (określonym przez drugi argument procedury `sgml`) jest zapisanie na wyjście pustego ciągu znaków. Jako akcja może być też podana referencja do anonimowej (tak jak w przypadku zdarzeń `'start'` i `'end'`) lub nazwanej procedury.

Procedura, do której referencja jest podawana w funkcji `sgml` otrzymuje przy wywołaniu dwa argumenty: dane związane ze zdarzeniem (w przypadku elementów – obiekt `SGMLS_Element`) oraz samo zdarzenie.

Dokonanie konwersji takiego prostego dokumentu np. na format LaTeX-a można osiągnąć przez uzupełnienie szkieletu:

```
# Document Handlers.
#
sgml('start', sub {
       output<<'EOH';
\documentclass{article}
\usepackage{polski}
\pagestyle{empty}
\begin{document}
EOH
     });
sgml('end', sub {
     output<<'EOF';
\end{document}
});

# Element Handlers.

# Element: CHAPTER
sgml('<CHAPTER>', "");
sgml('</CHAPTER>', "");

# Element: TITLE
sgml('<TITLE>', "\n\\chapter{");
sgml('</TITLE>', "}");

# Element: PARA
sgml('<PARA>', "");
sgml('</PARA>', "\n\n");

# Element: ITEMIZEDLIST
sgml('<ITEMIZEDLIST>',
     "\n\\begin{itemize}\n");
sgml('</ITEMIZEDLIST>',
     "\n\\end{itemize}\n");

# Element: LISTITEM
sgml('<LISTITEM>', "\\item\n");
sgml('</LISTITEM>', "");
```

Rzeczywistość nie jest zwykle jednak tak prosta. Prawdziwe dokumenty będą zawsze dużo bardziej skomplikowane. Przykład bardziej rzeczywistej (chociaż wciąż dosyć prostej) specyfikacji przetwarzania dokumentu można zobaczyć w dodatku. Jest ona przeznaczona do konwertowania prostych dokumentów w SGML-u do LaTeX-a.

Jade – implementacja języka stylów DSSSL. Jeśli chcemy używać szablonów DSSSL dla DTD DocBook, potrzebny będzie program `jade`, będący implementacją języka stylów DSSSL – czyli podzbioru standardu DSSSL. Autorem programu, podobnie jak pakietu SP, jest James Clark. Za pomocą `jade` można „produkować" dokumenty w formacie RTF, TeX, HTML, MIF (Frame Maker).

Podstawową zaletą używania DSSSL-a i programu `jade` do formatowania dokumentów jest możliwość uzyskania prawie identycznego formatowania w TeX-u i w RTF-ie. W pewnych okolicznościach możliwość zapisania dokumentu w formacie RTF, bez konieczności używania najpopularniejszego procesora tekstów na świecie, a nawet bez uruchamiania systemu „W", jest bardzo cenna.

docbook-stylesheets – szablony stylów DSSSL dla DTD DocBook. DSSSL-owy zestaw szablonów dla DTD DocBook wspominany był już parę razy. Jest to dzieło imponujące (przynajmniej dla mnie). W komplecie znajdują się dwa zestawy – jeden przeznaczony do produkowania wersji „drukowalnej" (RTF, Jadetex), a drugi do tworzenia wersji hipertekstowej (HTML). Szablony te są bardzo dobrze udokumentowane, elegancko sparametryzowane (plik `db-param.dsl`) i łatwo je dostosować do swoich potrzeb. Aby sformatować dokument z wykorzystaniem szablonów Walsha należy wywołać `jade` następująco:

`jade -t` ⟨*format wyjściowy*⟩ `-d` ⟨*szablon*⟩

gdzie format wyjściowy to `sgml` (jeśli chcemy przekształcić dokument na HTML), `rtf` albo `tex`. Szablon dla HTML-a jest inny niż szablon dla dokumentów drukowalnych. Zakładając, że mamy odpowiednio ustawione ścieżki oraz pliki katalogowe proces konwersji na HTML, RTF oraz TEX należy uruchomić następująco:

```
jade -t sgml -d html/docbook.dsl doc.sgml
jade -t tex -d print/docbook.dsl doc.sgml
jade -t rtf -d print/docbook.dsl doc.sgml
```

Wyjściowy plik TEX-owy należy przetworzyć programem `jadetex`, aby otrzymać drukowalną wersję dokumentu.

Szablony Walsha są elegancko sparametryzowane i łatwo modyfikowalne. Jak radzi sam autor, najlepszym sposobem dostosowania ich do swoich potrzeb jest utworzenie własnego pliku sterującego (ang. *driver file*) zawierającego ustawienia lokalne i odwołanie do oryginalnego szablonu.

Szkielet takiego pliku sterującego wygląda następująco:

```
<!DOCTYPE style-sheet PUBLIC
"-//James Clark//DTD DSSSL Style Sheet//EN"
[
<!ENTITY dbstyle SYSTEM "docbook.dsl"
                                CDATA DSSSL>
]>
<style-sheet>
<style-specificationuse="docbook">
<style-specification-body>

;; tutaj umieszczamy własne ustawienia np.

(define %line-spacing-factor% 1.2)
(define %visual-acuity% "presbyopic")
(define %paper-type% "A4")
(define %body-start-indent% 2pi)

(element foreignphrase
  (let* ((nd (current-node))
         (language (attribute-string
                       "lang" nd)))
    (if language
        (make sequence
          (literal language)
          (literal " ")
          ($italic-seq$)
          )
        ($italic-seq$)
        )
    )
  )

;; itd...

</style-specification-body>
</style-specification>
<external-specification id="docbook"
                     document="dbstyle">
</style-sheet>
```

Definicje umieszczone w tym pliku przysłaniają to co jest zdefiniowane we wczytywanym później szablonie. Wszystkie parametry szablonu, których wartość można zmienić są zebrane w pliku `dbparam.dsl` i skomentowane. Każdy z szablonów („drukowalny" i hipertekstowy) ma inne parametry, zapisane odpowiednio w podkatalogach `print` i `html` głównego katalogu szablonów. Wystarczy skopiować definicję z odpowiedniego pliku i zmienić wartość przypisywaną parametrowi tak jak w ostatnim przykładzie. Jak widać na przykładzie, można także definiować tutaj sposób obsługi elementów.

Jadetex – TEX-owy formater DSSSL. Do formatowania dokumentów z wykorzystaniem szablonów DSSSL konieczne są programy formatujące sterowane wyjściem z programu `jade`. Jedną z możliwości formatowania jest wykorzystanie TEX-owej implementacji formatera DSSSL-owego przygotowanego przez Sebastiana Rahtza. Jak pisze sam autor, Jadetex jest pakietem eksperymentalnym – mimo to w niektórych przypadkach jego użycie może być uzasadnione. Z powodzeniem używaliśmy tego narzędzia przy pisaniu książek na etapie przygotowywania „maszynopisu". Wykorzystując gotowe DTD (DocBook) oraz DSSSL-owe szablony stylów można tanim kosztem (mam na myśli nakład pracy konieczny do wyprodukowania „jakoś" sformatowanego dokumentu) formatować dokumenty przeznaczone do zredagowania i wprowadzenia poprawek. Przy formatowaniu z wykorzystaniem szablonów DSSSL kompletna specyfikacja formatowania jest określana na zewnątrz TEX-a, który jest wykorzystywany w zasadzie jedynie do połamania akapitów i nie da się tego zmienić, ponieważ generowane przez `jade` pliki TEX-owe absolutnie nie są przeznaczone do czytania czy modyfikowania przez ludzi.

Jakość składu pozostawia wiele do życzenia i w przypadku konieczności większej kontroli nad wyglądem dokumentu konieczne jest użycie innych narzędzi np. perla z biblioteką SGMLSpm.

Przy próbie użycia Jadetexa czeka nas jeszcze jedna niespodzianka. Oczywiście po przetworzeniu pliku nie będzie polskich liter. Mój przepis na ich uzyskanie jest być może nie całkiem elegancki, ale skuteczny. Przed wygenerowaniem formatu dla Jadetexa modyfikuję plik źródłowy w następujący sposób:

- Wyrzucam wywołanie pakietu `inputenc` z kodowaniem wejściowym `dsssl`, ponieważ koliduje ono z kodowaniem polskich liter.
- Zmieniam wywołanie pakietu `babel` wyrzucając język francuski i niemiecki.
- Dodaję wywołanie pakietu `polski`.

Transformacji polskich znaków dokonuję używając tablic `tcx`. W przypadku używania implementacji TEX-a, która nie ma możliwości używania mechanizmu przekodowywania znaków, można dołączyć pakiet `inputenc` z opcją `latin2` (lub inną w zależności od używanego kodowania polskich znaków).

Pewną niedogodnością Jadetexa są fonty, jakich można używać do składu dokumentu. W Jadetexu używane są fonty w układzie T1, który nie jest całkiem zgodny z OT4. Definicje niektórych znaków (np. pauzy), są „zaszyte” w trakcie generowania formatu na pozycjach z układu T1, więc nie bardzo da się używać np. fontu QuasiTimes. Jednak ponieważ Jadetexa nie używam nigdy do składu tekstu w ostatecznej postaci, to nie jest to (przynajmniej dla mnie) poważny problem.

sgmltools. Pisząc o formatowaniu dokumentów SGML-owych nie sposób nie wspomnieć od pakiecie `sgmltools`. Pakiet ten jest wynikiem rozwoju używanego przez długi czas w projekcie dokumentowania systemu Linux pakietu `linuxdoc-sgml`. Pierwotnie wykorzystywane było tam DTD QWERTZ – proste DTD definiujące w języku SGML prawie dokładne odpowiedniki standardowych LATEX-owych typów dokumentów. Do formatowania wykorzystywany był specjalny program oraz zestaw skryptów. Obecnie pakiet ten jest intensywnie reorganizowany: zmieniono DTD na DocBook, do formatowania wykorzystywany jest program `jade` oraz szablony Walsha i Jadetex. Wszystko jest sterowane za pomocą skryptów w pythonie. Całość wygląda imponująco i docelowo może powstać z tego całkiem sensowna maszyneria do formatowania dokumentacji.

Moje doświadczenia z tym pakietem są niewielkie, chociaż większości narzędzi wchodzących w jego skład używam już od dłuższego czasu. Być może dlatego wrażenia z próby użycia były nie najlepsze. Prawdę mówiąc wolę używać wszystkiego „świadomie”, a do zarządzania procesem formatowania używać programu `make`.

Rysunki, tabele i wzory w SGML-u

Istotnymi elementami dokumentów są ilustracje, tabele oraz formuły matematyczne. Także tutaj nie podam gotowego przepisu na radzenie sobie ze wszystkimi problemami jakie mogą pojawiać się w przypadku konieczności umieszczenia takich obiektów w dokumentach, a tylko kilka uwag związanych z tą tematyką.

Grafika. W SGML-u, podobnie jak w przypadku TEX-a, gdzie obsługa grafiki jest realizowana „na zewnątrz” przez sterowniki (`dvips`, `xdvi`, itd.), kwestia grafiki nie jest objęta standardem. Możliwe jest zaznaczenie, że w danym miejscu dokumentu znajduje się ilustracja. Może ona być nawet fizycznie dołączona do dokumentu, ale jej interpretacja musi być zrealizowana przez system używany do prezentacji. W SGML-u istnieje jedynie mechanizm definiowania tzw. notacji – określających zewnętrzne, mniej lub bardziej powszechnie używane standardy zapisywania informacji graficznej. Różnym rodzajom notacji (która nie musi tak naprawdę koniecznie dotyczyć grafiki sensu stricto) przypisane są identyfikatory publiczne (jeśli jest dostępna oficjalna definicja formatu) albo systemowe. Wszystkie notacje (formaty), do których odwołujemy się w dokumencie SGML-owym muszą być zadeklarowane w DTD albo w nagłówku dokumentu (w podzbiorze deklaracji typu dokumentu). Oto kilka przykładów notacji przewidzianych do użycia w DTD DocBook:

```
<!NOTATION EPS PUBLIC
"+//ISBN 0-201-18127-4::Adobe//NOTATION
PostScript Language Ref. Manual//EN">

<!NOTATION GIF87a PUBLIC
"-//CompuServe//NOTATION Graphics
Interchange Format 87a//EN">

<!NOTATION GIF89a PUBLIC
"-//CompuServe//NOTATION Graphics
Interchange Format 89a//EN">
```

```
<graphic fileref="gustlogo.eps"
format="eps"></graphic>
```

Rys. 1. Dołączanie grafiki do dokumentu SGML-owego

Dołączenie grafiki do dokumentu SGML-owego zakodowanego z wykorzystaniem DocBooka jest możliwe przez wykorzystanie elementu `<graphic>`. Elementowi temu można podać atrybuty określające format danych graficznych, które chcemy dołączyć w miejscu użycia elementu oraz miejsce gdzie przechowywane są te dane (najczęściej oczywiście w zewnętrznych plikach). Tak właśnie zostało dołączone do tego tekstu logo GUST-u zapisane w pliku postscriptowym `gustlogo.eps`. Cała odpowiedzialność za właściwą intepretację grafiki jest pozostawiona systemowi formatującemu. Co więcej, różne sposoby prezentacji mogą wymagać różnego przygotowania ilustracji – jeśli z tego samego dokumentu źródłowego zakodowanego w SGML-u będzie trzeba przygotować kilka wersji prezentacyjnych (np. PS – do druku, HTML i PDF do sieci), to wszystkie pliki graficzne muszą być przygotowane w odpowiednim formacie i z uwzględnieniem ograniczeń i możliwości danego sposobu prezentacji.

Jeśli przewidujemy zawczasu możliwość różnorodnej prezentacji (a jeśli byłoby inaczej, to właściwie – po co nam SGML), to powinniśmy sparametryzować dokument tak, żeby w czasie przetwarzania do pewnego formatu prezentacyjnego dołączane były odpowiednie wersje ilustracji. Do parametryzacji najwygodniej jest użyć parametryzujących encji (tak jak w przykładzie definiowania logo TEX-owego). Konstrukcja taka mogłaby wyglądać więc tak:

```
<![%TEX[
<graphic fileref="gustlogo.eps"
format="eps"></graphic>
]]>
<![%HTML[
<graphic fileref="gustlogo.gif"
format="gif89a"></graphic>
]]>
<![%PDF[
<graphic fileref="gustlogo.pdf"
format="pdf"></graphic>
]]>
```

Taki sposób jest nieco niewygodny, bo konieczne jest parametryzowanie wszystkich ilustracji w tekście dokumentu. W przypadku używania DocBooka można to załatwić bardziej elegancko. Elementowi `<graphic>` można w tym DTD podać atrybut `entityref`, który opisuje zewnętrzną encję (typowo plik), zawierającą ilustrację. Jak wiemy, definicje encji wykorzystywanych w dokumencie możemy umieścić w podzbiorze deklaracji typu dokumentu i dokonać parametryzacji właśnie tam. Taki sposób ma dodatkową zaletę, że w jednym miejscu mamy zgromadzone nazwy wszystkich plików dodatkowych niezbędnych do właściwej prezentacji dokumentu.

```
<!DOCTYPE article ... [
<!ENTITY % HTML "IGNORE">
<![%HTML[
<!ENTITY % TEX "IGNORE">
]]>
<!ENTITY % TEX "INCLUDE">
<![%TEX[
<!ENTITY logo SYSTEM "gustlogo.eps"
NDATA eps>
...
-- definicje innych encji zewnętrzych
   zawierających grafikę --
]]>
<![%HTML[
<!ENTITY logo SYSTEM "gustlogo.gif"
NDATA gif89a>
...
]]>
]
...
<graphic entityref="logo"></graphic>
```

W definicji encji zewnętrznej podajemy, że plik ten nie zawiera danych SGML-owych (słówko `NDATA`) oraz określamy format, w którym są zapisane dane. Jak pamiętamy, encji zewnętrznych można używać także do podziału dokumentu SGML-owego na mniejsze fragmenty – tam nie było słowa `NDATA` i identyfikatora notacji.

Tabele. Bardziej skomplikowane problemy niż z grafiką występują w przypadku tabel. Oczywiście można potraktować tabele jako grafikę i zastosować dokładnie to samo podejście. W przypadku grafiki dosyć łatwo się pogodzić z faktem, że dla różnych form prezentacji trzeba przygotować oddzielne wersje ilustracji, no bo taka jest natura grafiki i nie bardzo da się cokolwiek na to poradzić. Tabele jednak, jak każdy natychmiast zauważy, są niejako z założenia idealnie strukturalne (tabela składa się z wierszy, które składają się z rubryk) i jako takie potencjalnie bardzo dobrze nadają się do zakodowania w SGML-u. W przypadku prostych „przyzwoicie obramkowanych" tabel, nie zawierających rubryk rozciągających się na wiele wierszy lub kolumn i nie zawierających dłuższych fragmentów tekstu sprawa jest rzeczywiście prosta – łatwo też takie tabele formatować w dowolnym systemie (rys. 2 – sposób konwersji identyczny jak w przykładzie w dodatku – elementy od `ROW` do `TGROUP`). Jednak opracowanie ogólnego modelu, który dałoby się zastosować do dowolnych tabel nie jest proste. Należy uwzględnić różny sposób wyrównywania materiału w kolumnach, możliwość definiowania rubryk obejmujących wiele wierszy i kolumn, wymyślić sposób definiowania ramek itd. W chwili obecnej istnieją co prawda dwa niezłe SGML-owe modele tabel: CALS, opracowany w Departamencie Obrony Stanów Zjednoczonych oraz HTML3, opracowany w W3 Konsorcjum. Model CALS jest wykorzystywany w DTD DocBook. Opracowanie modelu struktury tabeli jest jednak tylko połową sukcesu – jeśli chcemy dokument prezentować w różnych formatach, to może okazać się, że używany model struktury tabel (zwłaszcza, jeśli tabele nie są proste) nie badzo pasuje do możliwości systemu formatującego.

	HTML	PDF	PS
Hipertekstowy	Tak	Tak	Nie
Binarny	Nie	Tak	Tak
Kompresja	Nie	Tak	Tak

Rys. 2. Prosta tabela konwertowana automatycznie

W praktyce realizowałem już projekty, w których znakowane w SGML-u dosyć skomplikowane tabele (z cieniowaniem fragmentów, wielokolumnowymi i wielowierszowymi rubrykami) z powodzeniem automatycznie konwertowałem do TEX-a (tabele były składane za pomocą pakietu `tap`). Było to jednak dosyć skomplikowane i mimo wszystko rozwiązane w mało ogólny sposób.

Formuły matematyczne. Największy problem jest z wzorami matematycznymi. Problemy są oczywiście spowodowane w tym przypadku także samą naturą notacji matematycznej jako takiej. Z założenia jest to notacja wieloznaczna – te same symbole są używane na oznaczenie różnych logicznie rzeczy. Z drugiej strony często jest tak, że te różne symbole oznaczają dokładnie to samo – np. pochodna funkcji $f(x)$ może być zapisana jako:

$$f'(x) \quad \text{albo} \quad df/dx \quad \text{albo} \quad \frac{df}{dx}$$

Notacja matematyczna ma też podwójną naturę – z jednej strony znaczeniową: w zasadzie wszystko jedno czy zapiszemy `A_i` czy A_i i tak wiemy o co chodzi, ale na papierze ma pojawić się właśnie A z pomniejszoną i obniżoną literą i, czyli istotna, mimo iż umowna, jest forma graficzna (jeszcze bardziej widać to w przypadku ułamków piętrowych).

Jednym ze sposobów poradzenia sobie z matematyką może być podejście zastosowane przeze mnie powyżej – wzory są tutaj kodowane w notacji TEX-owej – wykorzystywany jest mechanizm umożliwiający umieszczanie w dokumencie SGML-owym tekstu, który ma być interpretowany przez system formatujący z jawnym określeniem jaka notacja została użyta:

```
<inlineequation>
<graphic notation="tex">
{df\over dx}
</graphic>
</inlineequation>
```

Są oczywiście prowadzone prace nad opracowaniem języka znaczników do znakowania wzorów matematycznych. Opracowany jest nawet język MathML (z wykorzystaniem standardu XML), którego założenia były bardzo ambitne, bo zdecydowano się opracować funkcjonalny odpowiednik tradycyjnej notacji matematycznej. Ze względu na podwójną, graficzno-logiczną naturę notacji matematycznej w języku MathML przewidziano dwa sposoby definiowania wzorów: wizualny i znaczeniowy. Dzięki zastosowaniu podejścia strukturalnego MathML potencjalnie (na razie) daje możliwość dokonywania automatycznych operacji na zapisanych w nim formułach – konwertowanie do i z istniejących formatów (np. TEX-owego), używanie w systemach obliczeń symbolicznych, prezentowanie w różny sposób – fonetycznie z wykorzystaniem syntezatorów mowy, na zwykłych tekstowych teminalach (np. VT100), graficznie – w tym także dla niewidomych z wykorzystaniem systemu Braille'a. Projekt bardzo ambitny i już dosyć mocno zaawansowany, ale jeszcze niestety nie trafił pod strzechy. Jednym z podstawowych założeń projektu było dopasowanie się do już istniejących rozwiązań, czyli możliwość konwersji lub generowania kodu MathML z wykorzystaniem narzędzi i edytorów obecnie używanych. I powoli wszystko rzeczywiście rozwija się w tę stronę: wsparcie dla MathML-a jest przewidziane w konwerterach `latex2html`, `tex4ht`. Powstają już także DSSSL-owe style przeznaczone do wizualizacji wzorów oznakowanych z wykorzystaniem języka MathML (na rys. 3 widać wzór zapisany poniżej w dwóch wariantach notacji MathML-owej). Na razie wszystko wskazuje, że dzięki rozpowszechnieniu języka MathML i narzędzi konwertujących (oby!) możliwe będzie pozostanie przy notacji, do której jako użytkownicy TEX-a przywykliśmy.

```
<!-- Notacja ,,wizualna'' -->

<mrow>
  <msqrt>
    <mi>&Delta;</mi>
  </msqrt>
  <mo>=</mo>
  <msqrt>
    <mrow>
      <msup>
        <mi>b</mi>
        <mn>2</mn>
      </msup>
      <mo>-</mo>
      <mrow>
        <mn>4</mn>
        <mo>&InvisibleTimes;</mo>
        <mi>a</mi>
        <mo>&InvisibleTimes;</mo>
        <mi>c</mi>
      </mrow>
    </mrow>
```

```
  </msqrt>
</mrow>

<!-- Notacja ,,znaczeniowa'' -->

<reln>
  <eq/>
  <apply>
    <root/>
    <ci>&Delta;</ci>
  </apply>
  <apply>
    <root/>
    <apply>
      <minus/>
      <apply>
        <power/>
        <ci>b</ci>
        <cn>2</cn>
      </apply>
      <apply>
        <times/>
        <cn>4</cn>
        <ci>a</ci>
        <ci>c</ci>
      </apply>
    </apply>
  </apply>
</reln>
```

$$\sqrt{\Delta} = \sqrt{b^2 - 4ac}$$

Rys. 3. MathML → DSSSL → Jadetex

Zamiast zakończenia

Na tym kończę te nieco nieuporządkowane uwagi dotyczące praktycznych aspektów pracy z SGML-em. Nie wiem czy opisane tutaj narzędzia oraz sposób przetwarzania dokumentów będzie odpowiadał innym. Ja w każdym razie już od dłuższego czasu w zasadzie wszystkie teksty (poza prezentacjami, chociaż już namawiano mnie do przygotowania SGML-owego interfejsu do prezentowanego w innym tekście w tych materiałach pakietu `pdfslides`) piszę w SGML-u wykorzystując DocBooka jako DTD i Emacsa jako edytora strukturalnego. Na potrzeby formatowania większości tekstów przygotowuję zmodyfikowaną wersję konwerterów wykorzystujących bibliotekę SGMLSpm. Mam też prawie gotowy skrypt do konwersji artykułu zakodowanego w SGML-u na format biuletynu GUST czy stylu `tugproc`. Za namową kolegów z GUST-u udostępnię na serwerze część swojego SGML-owego „dorobku”. Potencjalnych użytkowników proszę o wyrozumiałość oraz kontakt w razie wykorzystywania lub rozszerzania tych narzędzi.

Źródła informacji

Dokumentacja dotycząca SGML-a i okolic jest bardzo bogata. Bardzo bogaty (i wciąż uaktualniany) zestaw dokumentów dotyczących zagadnień tekstów strukturalnych jest dostępny na stronie OASIS (`http://www.oasis-open.org/cover/sgml-xml.html`). Istnieje także strona zawierająca materiały o SGML-u w języku polskim (`http://www.fuw.edu.pl/~ksiezyk/sgml.html`).

Literatura

[1] Charles F. Goldfarb: *The SGML Handbook*, Clanendon Press, 1995.

[2] Martin Bryan: *SGML An Author's Guide to the Standard Generalized Markup Language*, Addison-Wesley, 1995.

[3] Erik van Hervijnen: *Practical SGML*, Kluwer Academin Publisher, 1994.

[4] Eve Maler, Jeanne El Andaloussi: *Developing SGML DTDs. From Text to Model to Markup*, Prentice Hall PTR, 1996.

[5] Włodzimierz Wypych: *SGML – technologia opracowania dokumentów elektronicznych*, Nowe Media, październik 1997.

[6] ISO/IEC: *Document Style Semantics and Specification Language (DSSSL)*, ISO/IEC 1017:1996(E).

[7] Sebastian Rahtz: *The TEX backend for Jade and JadeTEX macros*, August 1998.

[8] David Megginson: *SGMLS.pm: a perl5 class library for handling output from SGMLS and NSGMLS parsers (version 1.03)*, 17 December 1996.

[9] David Megginson: *sgmlspl: a simple post-processor for SGMLS and NSGMLS parsers (for use with SGMLS.pm version 1.03)*, 17 December 1996.

[10] Terry Allen, Eve Mahler: *Overview of the DocBook DTD*, Release 2.4.1, January 1997.

[11] Terry Allen, Eve Mahler: *User's Guide for the DocBook DTD*, Release 2.4.1, January 1997.

[12] Terry Allen, Eve Mahler, Norman Walsh: *Reference for the DocBook DTD*, Release 3.0, January 1997.

[13] Terry Allen, Eve Mahler: *Customizer's Guide for the DocBook DTD*, Release 2.4.1, January 1997.

[14] Patrick Ion, Robert Miner (eds.): *Mathematical Markup Language (MathML) 1.0 Specification*, W3C Recommendation, 07-April-1998.

[15] James Clark: *SP (* `http://www.jclark.com/sp/`*)*.

[16] James Clark: *Jade – James' DSSSL Engine (*`http://www.jclark.com/jade/`*)*.

[17] Norman Walsh: *The Modular DocBook Stylesheet*, 1998.

[18] David Carlisle: *MathML Files: DSSSL style sheet for MathML (*`http://www.nag.co.uk/projects/openmath/mml-files/`*)*, 1998.

Przykład specyfikacji przetwarzania dokumentu dla sgmlspl

```
########################################################################
# SGMLSPL script produced automatically by the script sgmlspl.pl
# Document Type: ARTICLE
# Edited by: Piotr Bolek
########################################################################

use SGMLS;                      # Use the SGMLS package.
use SGMLS::Output;              # Use stack-based output.
use SGMLS::Refs;

#
# Document Handlers.
#
sgml('start', sub {
  $Refs = new SGMLS::Refs('.latex.refs');
});
sgml('end', sub {
  $Refs->warn;
  #destroy $Refs;
});

#
# Element Handlers.
#
sgml('<ARTICLE>', "
\\documentclass[12pt]{article}
\\usepackage{d2l}
\\begin{document}
");
sgml('</ARTICLE>', "\\end{document}");

sgml('<ARTHEADER>', "");
sgml('</ARTHEADER>', "\\maketitle");

sgml('<TITLE>', sub {
    my $element=shift;
    my $parent=$element->parent->name;
    push_output 'string';

});
sgml('</TITLE>', sub {
    my $element=shift;
    my $t;

    $t=pop_output;
    $_=$element->parent->name;
  TITLE: {
      /ARTHEADER/ and do { $t="\n\\title{$t}\n";   last TITLE };
      /SECT1/     and do { $t="\n\\section{$t}\n";    last TITLE };
      /SECT2/     and do { $t="\n\\subsection{$t}\n"; last TITLE };
      /FIGURE/    and do { my $num = &counter($_);
                           $t="\n{\\smallskip\\centerline{Rys. $num. $t}}\n";
                           $figcap=$t; undef $t;
                           last TITLE };
      /BIBLIOENTRY/ and do { &push;
                             output $t; undef $t;
                             &getcontents($element);
                             last TITLE };
  }
```

```
        output $t;
    });

    sgml('<BIBLIOGRAPHY>', "\n\\begin{thebibliography}{99}\n");
    sgml('</BIBLIOGRAPHY>', "\\end{thebibliography}\n");

    sgml('<BIBLIOENTRY>', "");
    sgml('</BIBLIOENTRY>', sub {
           my $element=shift;
           my $id=$element->attribute('ID')->value;
           $id=$autoid++ unless $id;
           output "\n\\bibitem{$id} $authorgroup:\n" if $authorgroup;
           output "\n\\bibitem{$id} $firstname $othername $surname:\n" if $surname;
           output "\n\\bibitem{$id} $corpauthor:\n" if $corpauthor;
           output "{\\sl $title}";
           output ", $edition" if $edition;
           output ", $publisher" if $publisher;
           output ", $pubdate" if $pubdate;
           output ".\n";

           undef $authorgroup; undef $firstname;  undef $othername;
           undef $surname;     undef $title;      undef $publisher;
           undef $pubdate;     undef $corpauthor; undef $edition;
         });

    sgml('<AUTHOR>', "");
    sgml('</AUTHOR>', sub {
        my $element=shift;
        my $parent=$element->parent->name;

        if ($parent eq "AUTHORGROUP") {
          output ", " if ($auth);
          $auth++;
          output "$firstname $othername $surname";
            undef $firstname; undef $othername; undef $surname;
        }
        if ($parent eq "ARTHEADER") {
            output "\\author{$firstname $othername $surname}\n";
            undef $firstname; undef $othername; undef $surname;
        }
    });

    sgml('<AUTHORGROUP>', \&push);
    sgml('</AUTHORGROUP>', \&getcontents);

    sgml('<FIRSTNAME>', \&push);
    sgml('</FIRSTNAME>', \&getcontents);

    sgml('<CORPAUTHOR>', \&push);
    sgml('</CORPAUTHOR>', \&getcontents);

    sgml('<EDITION>', \&push);
    sgml('</EDITION>', \&getcontents);

    sgml('<PUBLISHER>', \&push);
    sgml('</PUBLISHER>', \&getcontents);

    sgml('<OTHERNAME>', \&push);
    sgml('</OTHERNAME>', \&getcontents);
```

```
sgml('<SURNAME>', \&push);
sgml('</SURNAME>', \&getcontents);

sgml('<PUBDATE>', \&push);
sgml('</PUBDATE>', \&getcontents);

sgml('<DATE>', \&push);
sgml('</DATE>', \&getcontents);

sgml('<ADDRESS>', \&push);
sgml('</ADDRESS>', \&getcontents);

sgml('<ABSTRACT>', "\n\\begin{abstract}\n");
sgml('</ABSTRACT>', "\\end{abstract}\n");

sgml('<CITY>', \&push);
sgml('</CITY>', \&getcontents);

sgml('<EMAIL>', \&push);
sgml('</EMAIL>', \&getcontents);

sgml('<PARA>', sub {
    my $element=shift;
    my $role=$element->attribute(ROLE)->value;
    my $parent=$element->parent->name;
    unless ($role eq "noindent" || $parent eq "FOOTNOTE") {
        output "\n";
    }
});
sgml('</PARA>', sub {
    my $element=shift;
    my $parent=$element->parent->name;
    unless ($parent eq "FOOTNOTE") {
        output "\n";
    }
});

sgml('<PROGRAMLISTING>', sub {
    my $element=shift;
    my $role=$element->attribute('ROLE')->value;
    output "\\begin{verbatim}\n";
});
sgml('</PROGRAMLISTING>', "\n\\end{verbatim}");

sgml('<LITERALLAYOUT>', sub {
    my $element=shift;
    my $role=$element->attribute('ROLE')->value;
    output "\\begin{verbatim}\n";
});
sgml('</LITERALLAYOUT>', "\n\\end{verbatim}\n");

sgml('<FIGURE>', sub {
    my $element=shift;
    my $float=$element->attribute('FLOAT')->value;
    my $id=$element->attribute('ID')->value;
    my $num=&stepcounter($element->name);

    $Refs->put($id, $num) if $id;
    output "\n\\begin{figure}\n";
    if ($float == 0) {
```

```
        output "[htb]";
    } elsif ($float == 1) {
        output "[t]";
    } elsif ($float == 2) {
        output "[b]";
    } else {
        output "[h]";
    }
});
sgml('</FIGURE>', sub {
    output "$figcap\n\\end{figure}\n\n"
});

sgml('<GRAPHIC>', sub {
    my $element=shift;
    my $fileref=$element->attribute(FILEREF)->value;
    my $format=$element->attribute(FORMAT)->value->name;
    my $width=$element->attribute('WIDTH')->value;

    if ($width=~ /^\d+$/) {
        $width=$width/100 . "\\hsize";
        $width="\\epsfxsize=$width\n";
        print STDERR "$width";
    }
    if ($format eq "EPS") {
        print "\\centerline{$width\\epsfbox{$fileref}}\n" if -e $fileref;
    }
});
sgml('</GRAPHIC>', "");

sgml('<INLINEEQUATION>', "\$");
sgml('</INLINEEQUATION>', "\$");

sgml('<EMPHASIS>', "\\emph{");
sgml('</EMPHASIS>', "}");

sgml('<FOOTNOTE>', sub{
    my $element=shift;
    my $num=&stepcounter($element->name);
    output("\\footnote{");
});
sgml('</FOOTNOTE>', "}");

sgml('<ITEMIZEDLIST>', "\\begin{itemize}\n");
sgml('</ITEMIZEDLIST>', "\\end{itemize}\n");

sgml('<ORDEREDLIST>', "\\begin{enumerate}\n");
sgml('</ORDEREDLIST>', "\\end{enumerate}\n");

sgml('<LISTITEM>', sub {
    my $element=shift;
    my $parent=$element->parent->name;
    if ($parent ne 'VARLISTENTRY') {
        output "\\item ";
    }
});
sgml('</LISTITEM>', "");

sgml('<VARIABLELIST>', "\\begin{description}\n");
sgml('</VARIABLELIST>', "\\end{description}\n");
```

```
sgml('<TERM>', "\\item{\\bfseries");
sgml('</TERM>', "} ");

sgml('<PARAMETER>', "<");
sgml('</PARAMETER>', ">");

sgml('<COMMAND>', "\verb|");
sgml('</COMMAND>', "|");

sgml('<MARKUP>', sub {
    my $element=shift;
    my $inverbatim=$element->within(LITERAL)
        || $element->within(PROGRAMLISTING);
    output "\\texttt{";
    output "<";
});
sgml('</MARKUP>', sub {
    my $element=shift;
    my $inverbatim=$element->within(LITERAL)
        || $element->within(PROGRAMLISTING);
    output ">}";
});

sgml('<LITERAL>', sub {
    my $element=shift;
    my $intitle=$element->within(TITLE);
    my $infoot=$element->within(FOOTNOTE);

    if ($intitle || $infoot) {
      output "\\texttt{";
    } else {
      output "\\verb+";
    }
});
sgml('</LITERAL>', sub {
    my $element=shift;
    my $intitle=$element->within(TITLE);
    my $infoot=$element->within(FOOTNOTE);

    if ($intitle || $infoot) {
      output "}";
    } else {
      output "+";
    }
});

sgml('<PARAMETER>', "<");
sgml('</PARAMETER>', ">");
sgml('<FOREIGNPHRASE>', "ang. {\\it ");
sgml('</FOREIGNPHRASE>', "}");

sgml('<XREF>', sub {
    my $element=shift;
    my $linkend=$element->attribute(LINKEND)->value;
    my $role=$element->attribute(ROLE)->value;
    my $value = $Refs->get($linkend) || "???";
    output($value);
});
sgml('</XREF>', "");
```

```
sgml('<ROW>', sub {
       $entry=0;
       output "\\-\n\\B|+";
     });
sgml('</ROW>', sub {
       output "\\E|\n";
     });

sgml('<ENTRY>', sub {
       my $element=shift;
       my $align=$element->attribute('ALIGN')->value;
       print "|" if $entry;
       output lc("\\$align\{") if $align;
       $entry++;
     });

sgml('</ENTRY>', sub {
       my $element=shift;
       my $align=$element->attribute('ALIGN')->value;
       output "\}" if $align;
     });

sgml('<TBODY>', "");
sgml('</TBODY>', "");

sgml('<THEAD>', "");
sgml('</THEAD>', "");

sgml('<INFORMALTABLE>', "\\hbox to \\hsize{\\hfill");
sgml('</INFORMALTABLE>', "\\hfill}");

# Element: TGROUP
sgml('<TGROUP>', sub {
       my $element=shift;
       my $cols=$element->attribute('COLS');
       output "%\\thistable{\\desiredwidth\\hsize}
\\begintable\\begintableformat & \\left \\endtableformat\n";
     });
sgml('</TGROUP>', sub {
       output "\\-\n\\endtable\n";
     });

#
# SDATA Handlers
#
sgml('|[tex]|', "\\TeX{}");
sgml('|[TeX]|', "\\TeX{}");
sgml('|[MetaPost]|', "MetaPost");
sgml('|[METAFONT]|', "\\MF{}");
sgml('|[Metafont]|', "\\MFsl{}");
sgml('|[deg   ]|', "\$^\\circ\$");
sgml('|[minus ]|', "\$-\$");
sgml('|[hellip]|', "\\ldots");
sgml('|[tilde ]|', "\\(\\tilde{}\\)");
sgml('|[bsol  ]|', "\\(\\backslash\\)");
sgml('|[num   ]|', "#");
sgml('|[rcub  ]|', "\\}");
sgml('|[lcub  ]|', "\\{");
sgml('|[mdash ]|', " --- ");
```

```
sgml('|[ndash ]|', " -- ");
sgml('|[amp   ]|', "\\&");
sgml('|[l2h]|', "\\LaTeX2HTML");
sgml('|[latex]|', "\\LaTeX{}");
sgml('|[latex2e]|', "\\LaTeXe");
#
# Default handlers (uncomment these if needed).  Right now, these are set
# up to gag on any unrecognised elements, sdata, processing-instructions,
# or entities.
#
# sgml('start_element',sub { die "Unknown element: " . $_[0]->name; });
# sgml('end_element','');
# sgml('cdata',sub { output $_[0]; });
# sgml('sdata',sub { die "Unknown SDATA: " . $_[0]; });
# sgml('pi',sub { die "Unknown processing instruction: " . $_[0]; });
# sgml('entity',sub { die "Unknown external entity: " . $_[0]->name; });
# sgml('start_subdoc',sub { die "Unknown subdoc entity: " . $_[0]->name; });
# sgml('end_subdoc','');
# sgml('conforming','');

sgml('re', sub {
       my ($data,$event) = @_;
       if ($event->element->name eq 'LITERAL') {
         output " ";
       } else {
         output "\n";
       }
     });

sgml('pi',sub {
    my ($instr) = @_;
    if ($instr eq "nl") {
        output "\\hfill\\break";
    } elsif ( $instr eq "shorter" ) {
        output "\\looseness=-1{}";
    }
});

sub push{
    my $element=shift;
    push_output 'string';
}

sub getcontents{
    my $element=shift;
    my $name=lc($element->name);
    $$name=pop_output;
}

sub stepcounter{
    my $name=shift;
    ++$count{$name};
}

sub counter{
    my $name=shift;
    $count{$name};
}

1;
```