TEI から見た SGML のはなし

豊島正之 / mtoyo@lit.hokudai.ac.jp

これは、情報処理語学文学研究会 (JALLC) 会報 12 号 (1992 年 12 月) に掲載し、その後、情報処理語学文学研究会会報累積版 (1996 年 7 月) の p.94-110 に整形掲載されたものを、新たに整形して掲げるものです。再整形以外は、(著者の所属を含め)「累積版」に一切変更を加えていません。　2000 年 8 月

1　TEI と SGML の関係
2　DTD による文書構造の分離と構造の予期
3　SGML の表現力の問題
4　可搬性 portability と文字コードの問題
5　推奨文献
6　引用文献
7　付:データ抽象

1　TEI と SGML の関係

研究用テキストデータ交換の為の標準化試案 TEI(Text Encoding Initiative) [1] は、SGML(ISO 8879) に載る形で提案されている (ACH/ACL/ALLC 1990/1992)。

1.1　TEI への批判と反発

TEI には、次の様な批判が寄せられている。(TEI-L mailing-list より)。

1. テキスト解釈の表現を、恰もテキストの内部構造・意図であるかの様に表現している
2. 文書構造を予見して (DTD) それに合せた記述を強制するのは、テキスト研究に制約を与える
3. 上下階層的 (hierarchical) ではない構造の記述が困難
4. 非継起的 (non-serial) な構造の記述が困難
5. markup したテキストは読みにくい・長い
6. テキストはあるがままの形で享受したい、変なタグとかは入れないで欲しい

1 は descriptive markup そのものの本質的な問題、2 は TEI の方針の問題、3 と 4 は SGML の問題、5 と 6 は単なる感情的な反発である。TEI は交換形式であって、享受の形式ではない。(誰も markup された推理小説など読みたくはない)。以下では、1〜4 を主として扱う。

1.2　SGML とは何 ではない か

[1] **[累積版補注]**　ここでは TEI-P2 の事を指す。この稿の後に、TEI-P3 が公表された。豊島「TEI-P3 に就て」(本累積版収録) を参照。尚、TEI は、素直に「てぃーいーあい」と読む。

- SGML は roff や LaTeX の様な清書用言語ではない。SGML 処理系は清書出力を出さない。
- SGML は、RTF の様なワープロテキストの交換用言語ではない。
- SGML は、PostScript の様なプリントイメージの交換用言語ではない。
- SGML はデータベース言語ではない。

SGML は、フラットな文書の中にある構造を、文書の実体 (文字列) からは分離して、構造自体をそれと明示 (markup) して記述する為の言語である。

お望みなら SGML で書かれた文書を更に LaTeX の様な処理系に通して清書させたり、データベースへの運び込み import をしたり、ワープロやエディタに読み込ませて WYSISYG で「見た通り」の扱いをしたりも出来るが、それらの「お化粧」処理は、いずれも SGML 処理系の関知するところではない。SGML は単に情報の構造的な切り分けだけの為にあるのである。

SGML が、こうした切り分け (markup) に`<section>` ··· `</section>` の様な TAG を用いるのはよく知られているが、タグ付けは SGML の中心部分ではなく、構造記述の目印に過ぎない。タグの無い SGML 文書も可能である。(SHORTREF cf. p.4)。

SGML は既に ISO 規格 (ISO 8879-1986) がある[2]。

尚、SGML という acronym は、今では Standard Generalized Markup Language などという、中々もっともらしいものが与えられているが、元来は IBM 上の GML で、「GML」の由来は、作成者 Charles Goldfarb、Edward Mosher、Raymond Lorie の頭文字をつなげただけ (AWK と同じ) である。

1.3 markup は「意図」か「効果」か

SGML の主張する descriptive markup は、テキストが出来上がる局面に伴って、その内部構造の反映として行われるもので、既に成立しているテキストの外見の投影・転記ではない。markup が異なっても (たまたま) 結果が同一に帰する事は十分あり得るが、それらは markup し分けねばならない。というより、見た目には同一に帰しているものをマークし分けてこそ、markup の意味がある。

結果・効果という出力側の markup ではなく、意図・構造という入力側の markup である descriptive markup は、本来テキストの成立と共にあるものである。従って、著者自身が markup しながら書く場合はともかく、それ以外の既存のテキストに就ては、恰もその成立に立合っているかの様に装って markup する訳で、つまり、markup する人の解釈の表現であるものを、テキストに内在する構造の反映の様に見せる事になる。(この点が、TEI に対しても出されている大きな疑問 [前述の 1] の一つである)。

所で、偶然同一に帰したものの区別が markup に求められる事は確かだが、逆に、必然的に同一であるものを同一と指摘する為の markup も必要である。ところが、テキスト中の特定部分をマークして、それと同じものが繰り返されるという syntax は、正規文法 (正規表現) では書けないし、SGML にもうまい方法が見当たらない。

2 DTD による文書構造の分離と構造の予期

SGML は、文書の構成を完全に階層的 hierarchical に捉え、文書の構造記述は DTD(document type definition)、文書本体はその個別例 instance として分離し、具体的な文書の内容・実体から文書の構造を切り離す事でデータ抽象を図っている。(「データ抽象」に就ては末尾の付説を参照)。

諸要素 element の包含関係等は構造記述として DTD に抽象的に記述され、諸要素中の具体的な文字列は instance として扱われる。このモデルでは、個々の文書の instance は、抽象的な文書構造への値の設定を行うだけで、これを役所の書式に穴埋め式で書いて行く事に例えた人があるが、正しい。

[2] **[累積版補注]** 本稿後、JIS X4151-1992「文書記述言語 SGML」も制定された。

この様に構造記述とその「値」(instance) としての実体を分離して置けば、共通の文書を一つに括ったり、ある構造を持つ文書を一括して他の構造へ変換したりというのは苦も無く行える。事務書類や、新しい構造・要素を付け加えつつ定期的にバージョンアップせねばならないマニュアル類等では、確かに生産性は飛躍的に向上するだろう。

2.1 DTD は予期に反した構造を排除する

しかし SGML の DTD のメリットは、この生産性云々よりも、寧ろ文書の適合性の検査にある。

DTD を予め作って置く事で、正当な文書の構造を予期し、その予期に相違する不備な文書は全て自動的に排除する、という機能の方がはるかに重要である。

1. DTD の例 : [Chares Goldfarb `memo.dtd` (sgmls 0.8 付属) を多少改変]
   (以下 `-- ... --` は注釈)

letter.dtd

```
<!ELEMENT   letter  O O (header,body)>
        --  letter という要素は header の次に body が来る --
<!ELEMENT   header  O O (date,from,to,subject,myref,yourref?)>
        --  header は date, from, ... の順で、yourref は無くても可 --
<!ELEMENT   to      O O (whom, address)>
        --  to は whom, address の順 --
<!ELEMENT   (date | from | whom | address)  O   O   (#PCDATA) >
        --  date, from, ... は何れも文字列データ (Parsed Char data)--
<!ELEMENT   (subject | yourref | myref | p) O   O   (#PCDATA) >
        --  同上 --
<!ELEMENT   body    O O (p+) >
        --  body は p が一つ以上繰り返されたもの --
<!ATTLIST   letter
        --  letter 中の特定要素が取り得る値のリストアップ --
    security    (staff | domestic | public) #REQUIRED
        --  security は staff/domestic/public のどれか。
            しかも、security の値指定は必須である --
    status      (final | draft) final
        --  status は final/draft どちらかの値しか取れない。
            status の値を指定しない時は final と見なす --
>
```

2. 正しい instance (手紙) の例

goodlet.sgm

```
<!DOCTYPE letter SYSTEM>    -- 上の letter.dtd を使うと宣言 --
<letter security=domestic>  -- きちんと security 指定をしている --
<header>
    <date> nov.29,1992 <from> mtoyo
    <to>
        <whom>furuso <address> c/o JALLC
    </to>                   -- to 要素の終わり --
    <subject> sorry, draft isn't ready
    <myref> 921129.mac.f00  -- (yourref が無いのは構わない) --
</header>                   -- header 要素の終わり --
<body> Please allow me 2 days more. thanks.
</letter>   -- letter 要素の終わり (body 要素は自動的に終わる) --
```

3. エラーの instance (手紙) の例

badlet.sgm

```
 1   <!DOCTYPE letter SYSTEM>    --  上の letter.dtd を使うと宣言 --
 2   <letter status=tentative>   --  security 指定が無い,
 3                                   status の値が final/draft 以外 --
 4   <header>
 5       <date> nov.29,1992 <from> mtoyo
 6       <to>
 7           <whom>furuso        --  address を書いてない --
 8       </to>
 9       <subject> sorry, draft is still incomplete
10       <myref> 921202.mac.f00
11   </header>
12   <body> 2 more days please !
13   </letter>
```

3 番目の例 (badlet.sgm) では、2 行目 <letter>に対する security レベルの設定を怠り、status の値が予め定められたものでなく、且つ相手のアドレスも書いてないから、SGML 処理系がエラーを出して、この手紙を受け付けない。会社は、無駄な・保安上不備な手紙を出さずに済む。

ワープロやデータベースソフトウェアには、この類のチェックをするものもあるが、チェック機能がそうしたアプリケーションの働きとしてではなく、文書自体に組込まれている点が DTD の最大の特色である。つまり、

どういうのが適格な文書か、文書自体が知っている

のである。即ち、DTD **は本質的に文書を制約する為に存在する。**

尚、DTD 自体は SGML の一つの適用例に過ぎず、DTD 批判を直ちに SGML への批判とするのは当たらないとする意見がある。c のポインタでマナーの悪いプログラムが書けるからといって c 自体が悪い訳では無いという理屈と同じで、一応もっともに見えるが、実は無意味な反論である。

DTD を用いない SGML 文書というのは、struct やポインタの無い c プログラム同様、実際の現場ではまずお目に掛かれない代物で、おもちゃ同然である。SGML 文書に取って DTD は実質的には必須の存在であり、DTD の問題は直ちに SGML の問題である。

2.2 DTD はフラットなテキストも予期の構造で解釈する

DTD が、予期に反した文書構造を排除する機能を持つ事は前述の通りだが、SGML には更に、markup されていないフラットな文書に勝手に markup を補い、予期した構造通りに強引に読み下すという、SHORTREF 機能がある。これによれば、タグ無しの文書からも構造が読み取れる。(実のところは、文書の文字列上の特徴をタグに置き換えているだけなのだが)。

letref.dtd

```
<!ELEMENT   letref  O O (header, body)>
<!ELEMENT   header  O O (from, to?, date?) >
            -- to や date は無くてもよい --
<!ELEMENT   (from | to | date)  O O (h)+ >
            -- from や to は h が一つ以上 (h は下に定義) --
<!ELEMENT   body    O O (p+) >
            -- body は p が一つ以上 --
<!ELEMENT   (h | p)     O O (#PCDATA) >
            -- h や p は文字列 --
<!ENTITY    beghead STARTTAG    "h">
            -- beghead は h の 開始タグの呼称 --
<!ENTITY    begpara STARTTAG    "p">
            -- begpara は p の 開始タグの呼称 --
<!ENTITY    begbody STARTTAG    "body">
            -- begbody は body の 開始タグの呼称 --
<!ENTITY    endtag  ENDTAG      "">
            -- endtag は、最も手近のエレメントの終了タグ --
<!SHORTREF  letmap
            "&#RS;B"        begpara
            --  行頭に空白があったら、begpara だと思う。
                &#RS は行頭、B は空白。(「;」は単なる区切り)。
                つまり「&#RS;B」は行頭の空白。--
            "&#RS"          beghead
            --  ただの行頭は、begpara だと思う。 --
            "&#RS;&#RE"     begbody
            --  空行は、begbody だと思う。 &#RE は改行 (行末) --
            "&#RE"          endtag
            --  改行は endtab だと思う。 --
>
<!USEMAP    letmap  letref>
            --  letref は letmap を使って解釈する --
```

1. header をフルに書いた例

let1.sgm

```
<!DOCTYPE letref SYSTEM> -- 上の letref.dtd を使う ---
this is from field
this is to field
this is date field
    text body
```

sgmls の出力

```
(LETREF (HEADER (FROM (H -this is from field)H
(H -this is to field)H)
(H -this is date field)H)FROM)HEADER
(BODY (P -text body)P)BODY)LETREF
```

2. header を省略して、空行で区切った例

let2.sgm

```
<!DOCTYPE letref SYSTEM> -- 上の letref.dtd を使う ---
this is from field
    text body 1
    text body 2
    text end
```

sgmls の出力

```
(LETREF (HEADER (FROM (H -this is from field)H)FROM)HEADER
(BODY (P -text body 1)P (P -text body 2)P (P -text end)P)
 BODY)LETREF
```

(空行が BODY の始まりと認識されている事が分かる)。

この様な文書構造の強制は、標準化の手段としては大変有効で、うるさい markup 抜きの「見た通りの」文書が作れるというメリットもある。(Goldfarb はこれを extended WYSIWYG と呼んでいるが ···)。しか

し、研究対象としてのテキストの表現法としては疑問無しとしない。(勿論 TEI の DTD は、こんな単純なものではなく、又そもそも SHORTREF は使わない事になっているが、使う刃物がどの程度切れるかは知って置くべきである)。

更には、TEI の様な文書表現規格が蔓延すると、思想表現自体に対する規制になるのではないかと危惧する向きもある。(Sampson,Geoffrey 1992)。実際、将来の役所には「その様なタグは無いのでそうした申請は出来ない」、「そういうタグを新設した前例は無い」と権柄づくの小役人が出て来るのではないかと心配になる。因みに、これに対して TEI の盟主 Susan Hockey が TEI mailing-list 上で行った反論は、「TEI は過去の文書を表現・交換する為のもので、今後の文書作成には関係ありません」という大はずしで、mailing-list 講読者をがっかりさせた。

3 SGML の表現力の問題

SGML は構造を記述するのみであり、意味を直接に与える事が無い。(従って、SGML は「プログラム言語」ではない – 意味を与えない「プログラム言語」など無い)。これは「A marked-up document is highly portable, but the meaning of it is anybody's guess」(Erik Naggum) と皮肉られる事がある。

どうも SGML には、構造の記述 (syntax) によって文書の意味 (semantics) は殆ど定まり、他の部分はそれぞれの instance が特定値を与えるだけ、という前提がある様である。いわば、個々の instance を「固有名詞」(各 element は「名詞」) 扱いしている訳で、確かにこの前提が成立する文書は沢山ある。典型的な例は戸籍で、そこに書込まれた文字列は、位置によってそれぞれ「本籍」、「筆頭者」、「妻」などの (戸籍行政上の) 意味を担うだけで、それ以上の情報 (「沖縄県」、「太郎」等) は、戸籍行政上は本質的な意味の差をもたらさない単なるラベルである。文献データベース、マニュアル等にも同様のものがあるが、しかし、全ての文書がこうだとは限らない。

3.1 hierarchical に表現出来ない情報

SGML の content model は、上下階層的な hierarchy しか表現出来ない。(CONCUR という指定で、二つ以上の DTD を共存させる事は可能だが、依然、一 DTD 当たり一構造である – しかも CONCUR を実装した処理系は少ない)。

SUBDOC という指定で文書が文書を含む事も出来るが、これも依然 hierarchical である。しかも、この実現も相当困難らしく、sgmls v1.0 は漸くこれを実装したものの、exec で自分自身を起動し直すという大胆な方法を取ったので動作が不安定で、メモリを壊す事がある。

しかし、実際の文献には、hierarchy を無視した現象はいくらも起こる。掛け詞、一語の中に生じた異文・欠損 (虫損)、複数の文・パラグラフにわたる傍線・抹消・移動指示、韻文中のの複数行にまたがる散文引用 (eg. 戯曲)、セリフ (韻文) の途中への他者の脇ゼリフの割り込み、渡りゼリフ (cf. 落語「きゃいのう」)、etc.。これは、テキスト中に杭の様に目印を打って行き、それを参照するという方法である程度切り抜けられる。

```
  X  now  is  the         for
  Y       is      time    for all
  Z           the         for all good

I can do this:
  <x start id=x1>now <y start id=y1>is<y end>
  <z start id=z1>the<z end>
  <x end><y start refid=y1>time<x start refid=x1>
  <z start refid=z1>for all<y end> good<z end>
Not the best, but it does meet the challenge, I think.

(usenet の記事よりの引用)
From: DRMACRO@RALVM13.VNET.IBM.COM ("Dr. "Eliot Kimber" Macro")
Newsgroups: comp.text.sgml
Subject: Representing Overlapping Data Structures in SGML,Round 2
Message-ID: <9207231959.AA06068@ucbvax.Berkeley.EDU>
Date: 23 Jul 92 19:44:26 GMT
```

しかし、こうした <x start refid=x1>などといった attribute(<!ATTLIST>) の濫用は、結局文書の構造を不明瞭にする。attribute の内容を SGML は検査しないからである。<x start refid=x3>等と存在しない refid を指定しても、そのまま通って仕舞う。TEI P2, chapter 34 (Base Tag Set for Transcriptions of Spoken Text) は、この種の「杭打ち」方式での、相互にオーバラップする会話の記録の定式化を提案しているが、もしこれを採用するなら、単なる SGML パーザでは不十分で、attribute の中身に立ち入った validator が必須となるに違いない[3]。

更に、SGML の本質的な制約として、attribute は element にしか振れない。element の一部 (eg. 部分的な振り仮名) や複数 element にまたがる attribute(eg. 熟字訓の振り仮名) は、それぞれを element として定義し直すしかない。しかし、attribute を振る為に element を「切り直す」というのは、本末転倒ではなかろうか? 殊に虫損などの場合、それが有意味な element になる保障は全く無いが、有意味でない element を DTD に記述するなら、殆ど DTD の意味は失せて仕舞う。

3.2 serial に表現出来ない情報

serial でないテキストとしては、漢文の反読が典型である。

有朋自遠方來　　[朋、遠方自リ來タル有リと読む場合]

を

```
<lex seq=Large2>有</lex><lex seq=0>朋<lex seq=2>
自<lex seq=1>遠方<lex seq=Large1>來
```

と、「杭打ち」方式で番号付けして見ても、SGML が seq をチェックして反読して呉れる訳では無く、例え二箇所に同じ seq を付けたとしても、そのまま通って仕舞うから、これは markup とは呼びにくい。

尚、念の為に言うと、本来の「杭打ち」は EMPTY ELEMENT を用いるもの (典型的には丁付け) で、上記のタグは本当は「杭打ち」ではない。

4 可搬性 portability と文字コードの問題

4.1 二つの character set

文字コードには交換用と処理用を区別すべきである。交換用は通信線の上に乗せる為のもの、処理用は通信を前提としないものである。IBM 系の EBCDIC を交換用に使うという話しは (IBM 同士は別として) 余り聞かないし、JIS X0208(「情報交換用漢字符号」) の 7 ビットコード系を処理用に使うという話しも聞

[3] **[累積版補注]** TEI-P3 section 31(Multiple Hierarchies) は、こうした「杭打ち」方式を最終提案とし、validator の必要性を認めている。但し、TEI comformant な validator の存在は、今 (1996 年 5 月) に至るまで知られていない。尚、Barnard & Burnard(1995) に TEI-P3 の立場の説明があるので、併せて参照。

かない。しかし、往々にして両者が混同され、例えば純に処理用コードとして設計された Unicode [4]に対して、XON/XOFF が通らないから駄目、等という見当外れの評 (Epnews、1991、非署名稿) が現れたりしている。テキストデータ交換の為には、現状ではコード交換の国際規格 ISO2022 によるしか無いが、文書規格は必然的にコード系に関与するので、そこに問題が生じている。

4.2 SGML 宣言での文字コードの扱いは二重ポインタ

SGML は、文字コードの問題になるべく立ち入らない為に、文字コードを (も) データ抽象して、文書本体の文字コードを処理系非依存にしている。当該文書中で用いる文字コードは、文書先頭の SGML declaration 中に宣言する。

```
SGML declaration
    document character set
    concrete syntax : SYNTAX : syntax-reference character set
    ...
```

ここで

document char set (以下 DCS)　ファイルが書かれている実際の文字コード
syntax-reference char set(SCS)　SGML 文書中で参照する文字コード

SGML 文書中で文字コードに言及する際は、必ず SCS により、DCS にはよらない。つまり、SGML 文書で文字の序列番号 (所謂文字のコード値) に言及すれば、それは SCS の中での序列であって、DCS のそれではない。

例えば、SGML には、c の #define の様なマクロの一種 <!ENTITY > があり

```
<!ENTITY  ATMARK    "&#64" >   -- '@'(decimal 64 == 0x40) --
```

の様に、文字コード値が書ける。(これは TEI で頻用される)。ここに指定するコードは SCS のコードであり、DCS のそれではない。もし

```
<!ENTITY  ATMARK    "@" >
```

と LITERAL で書いて仕舞うと、(LITERAL は DCS そのものだから) これは DCS を指示した事になり、portability を損なう。

尚、やりたければ

DCS　ASCII
SCS　EBCDIC

等という事も自由ではあるが、この場合

```
<!ENTITY  ATMARK    "&#64" >
```

は、もはや '@'(ASCII 64 番) ではなく、EBCDIC での 64 番目の文字、即ち「空白」である。

つまり、SGML 文書のコード参照は、

```
typedef SCS *DCS;
DCS         *string;          (即ち  SCS **string;)
```

という「二重ポインタ」になっている。(cf. Macintosh のメモリ管理も同様)。

4.3 SGML 宣言の「クレタのパラドクス」

上に見た様に、SGML 文書の先頭には「私は ASCII です」とか書いてある訳である。しかし素朴な疑問として、事前にそれが ASCII である事を知らずに、その「私は ASCII です」自体が、どうして読めるんだろう、という問題がある。

[4] **[累積版補注]** ISO 10646 と現在の Unicode は別。

勿論、これは読めない。つまり、SGML 宣言の DCS 指定は、機械処理の為のものではなく、事前にプリントアウトなどの形で人間が読む為のものである。

ISO8879 [172](Goldfarb p.452) NOTE - It is recognized that the recipient of a document must be able to translate it to his system character set *before* the document can be processed by machine. ··· a human-readable copy of the SGML declaration will provide sufficient information.(強調豊島)

そう割り切ると、今度は別の問題が生ずる。即ち、「私は EBCDIC です」と宣言してある文書が通信線を伝ってやって来たら (かなりの確率で)ASCII/JISCII に変換されてしまうだろうという事である。つまり、SGML 宣言の「私は ... です」という自己言及命題：-) は、「クレタのパラドクス」の一種である。(今、そのファイルが現に用いているコードを、仮に「ACS : actual character set」と呼んで置く。)

4.4 SGML 文書での code extension

結局 SGML 文書のコードは、実は三重ポインタな訳で、

SCS ← DCS ↔ ACS

という対応関係にある。

従って、DCS と ACS は、相互に recoverable (送って受け直したら元に戻る) な交換が可能でなければならない。しかし、現行の大型機の漢字コード等では、この前提は全然成立しない。

[漢字 OUT]1 BYTE[漢字 IN] 2バイト [漢字 OUT]1[漢字 IN] 又 [漢字 IN] 2 [漢字 OUT]

を送って受け直すと

1 BYTE[漢字 IN] 2バイト [漢字 OUT]1[漢字 IN] 又2 [漢字 OUT]

になるかも知れないからである。

こういう状況で、SGML 文書中でエスケープシーケンス等を含む code extension (複数のコード表の取り替え) を行うと、全く portablity が損なわれる。つまり、portable にしようという前提の code extension が、仇になるのである。

従って、SGML の SCS を用いた code extension は行えない。実際、ISO8879 Annex E に code extension の例が出てはいるものの、単に ESC シーケンスを無視するだけの事で、SGML の規格自体は、厳密にいうと syntax での code extension を許していない様に読める。

4.42 code extension: Techniques for including in documents the coded representations of characters that are *not* in the document character set .

4.43 code set: A set of bit combinations of *equal size*, ordered by their numeric values, which must be consecutive.[強調豊島]

こうした SGML の文字コードモデルの下では、ISO2022 の様に G0〜G3 バッファを左右に呼出すモデルではなく、一直線の「シームレス」なコード系を仮想的に SCS に指定して仕舞う方がよいのではなかろうか。(JIS の「日本語 SGML」規格はそうしている[5])。SGML の SCS の上では、全ての文字のビット長は同一視 [4.43] され、どの文字も「一文字」として平等に扱える。これなら ACS と DCS の相互の厳密な recoverability は不要で、通常の通信ラインのコード変換が用い得るので好都合である。SCS の上で code extension を行うと、通常の通信ラインのコード変換は全て抑制せねばならないが、そんな事をする位なら、原ファイルをそのまま ISH や uuencode で送って仕舞った方がずっと楽である。

[5] **[累積版補注]** 本稿後に制定された JIS X4151-1992 「文書記述言語 SGML」は、文字コード関連は、EUC 方式の「表 9」を黙って追加したのみで、他は全て規格外の「解説」に押しやっている。これは、日本版規格としての責任を放棄した様なものである。

尚、Martin Bryan「SGML 入門」の 4.3(邦訳本 p.86-) は、code extension と本質的に無関係なので、誤解無き様。(この辺り、この書の記述は甚だ混乱していて、著者の理解の程を疑わせる)。

因みに M.Bryan は、usenet の[6]`comp.text.sgml` で、ISO646 IRV(ほぼ ASCII) 以外のコードの SGML 文書は規格合致でないと主張して購読者を唖然とさせた (`ref:1992May20.070618.25044@falch.no`) が、直ちにそれに同調したのが TEI の大立物 C.M.Sperberg-McQueen で、「そんな事では、ASCII 以外を使うマシンでは SGML 文書は読めない事になって仕舞う」というもっともな反論に対して

```
Yes ? So what ?  (ref:92142.195214U35395@uicvm.uic.edu)
```

と冷たく答えた [後に訂正] のは、記憶に新しい[7]。

4.5 transliteration と code extension

過激な選択としては、SGML 文書中では ISO646 IRV(ほぼ ASCII) しか用いず、それから外れる文字は全部 `&#2122` の様に ENTITY で示す (全面 transliteration ーというか 16 進ダンプ) という手がある。(TEI 初版では、実質この選択しか無かった)。勿論 SGML は「交換形式」であるので、これもあり得る選択であるが、SGML 文書には「human-readable」という大前提があるので、これは採れないだろう。(まぁ確かに「human-readable」には違い無いが、これでいいなら ISH だって human-readable である)。

従って、SCS の抽象性を生かした仮想コード系を作って処理し、code extension は DCS と ACS の間の通常のコード変換で処理するというのが現実的な選択になるだろう。Unicode/ISO10646 の様な国際共通コード系が一般的になれば、常に DCS==ACS になるので、尚更好都合である。

5 推奨文献

SGML に就て最初に読むべきは、ISO 規格の付録 A「Introduction to generalized markup」と付録 B「Basic concepts」[8]、更に Goldfarb(1990) の第二部「A structured overview of SGML」である。

Goldfarb(1990) は ISO 規格全文を含んでいるから、つまり Goldfarb(1990) さえあれば、入門には十分である。

SGML は、どういう訳か、まとまった文献の異様に少ない言語で[9]、他には Bryan (1988)、Herwijnen(1990) 位しか無いが、Bryan(1988) は記述が全然整理されておらず、無関係なものが突如割り込んだりして分かりづらさが尋常でないのみならず、そもそもどう見ても誤解としか思えない様な記述も散見し、全くお薦め出来ない。原題「Author's guide」は僭称に近いし、邦訳題「SGML 入門」は不適切。

Herwijnen(1990、邦訳ありと聞くも未見) は、面白い読み物だが、SGML の応用例に就ての本ではあっても、これを使って SGML を学ぼうとする人の為の本ではない。

Goldfarb(1990) は必備の文献である。その権威たるや、SGML の規格原文の引用すら (ISO 規格書ではなく) 慣習的に本書のページ数で引用される程で、当面はこれ一つで必要十分。

この他に、usenet のニューズグループ `comp.text.sgml` や、JUNET 等を経由して配送されている[10]電子メール上の会議 (メーリングリスト) TEI-L (購読申込先は `tei-L-request@uicvm.cc.uic.edu`)、corpora (`corpora-request@hd.uib.no`)、corpist (`ksonoda@ilcs.hokudai.ac.jp`) [11]が貴重な情報源である。

---

[6] **[累積版補注]** 現在では「internet 上の」。

[7] **[累積版補注]** 現に、TEI-P3 最終版でも、日本語などの 2 バイト文字は無視されたままである。

[8] **[累積版補注]** この二つには、JIS X4151-1992 に邦訳［というより英文解釈］がある。

[9] **[累積版補注]** その後にいくらか参考書が増えた。豊島「TEI-P3 に就て」(本累積版収録) 文献欄を参照。

[10] **[累積版補注]** 現時点 (1996 年 5 月) では、これらはいずれもインターネット上で配信されている。尚、JUNET は、1992 年に JUNET 協会となった後、1994 年 10 月に解散した。

[11] **[累積版補注]** これらは、あくまで現時点 (1996 年 5 月) でのアドレスなので注意。

6 引用文献

1.	ACH/ACL/ALLC	1990	TEI P1 (Guidelines for the encoding and interchange of machine-readable texts)/circulated draft
2.	ACH/ACL/ALLC	1992	TEI P2 /incomplete circulated draft
3.	Barnard,David & Burnard, Lou, et al.	1995	Hierarchical Encoding of text: technical problems and SGML solutions / Ide,Nancy & Veronis, Jean (eds.) (1995) *The Text Encoding Initiativesl* – Computers and the Humanities, 29-1/2/3 (special issue) [累積版追加]
4.	Bryan,Martin	1988	An author's guide to the SGML(邦訳「SGML 入門」、アスキー)
5.	EPnews(無署名)	1991	月刊「スーパーアスキー」1991-5;(P.156-)
6.	Goldfarb,Charles	1990	The SGML handbook(Oxford UP)
7.	ISO646	1983	Information processing - ISO 7-bit coded characterset for information interchange [1991 に改訂]
8.	ISO2022	1986	Information processing - ISO 7-bit and 8-bit coded charactersets – code extension techniques [1994 に改訂]
9.	ISO8879	1986	Information processing - text and office systems - standard generalized markup language
10.	Sampson,Geoffrey	1992	Writing within bounds / Times Literary Supplement Apr17-1992,p.14
11.	Sperberg-McQueen,C.M.	1991	Text in the electronic age : textual study and text encoding, with examples from medieval texts / Literary and linguistic computing 6-1(34-46)

7 付:データ抽象

データ抽象 data abstraction とは、データを、その直接の表現からは自由に、その本質に忠実に扱おうという主張である。

プログラミングの世界では、一貫してデータ抽象が試みられて来た。例えば、整数 (きりのいい正確な値) と実数 (近似値) では計算方法が異なるコンピュータは多く、太古のプログラム言語 FORTRAN では、実数と整数の相互直接代入すら禁止で

```
P = REAL(I)
```
(整数型 I を実数型に変換して実数型変数 P に代入)

と、人間が型変換を明示してやらねばエラーになった。

現在でも、80x86 系 CPU では、実数演算は専用プロセッサ (80x87) を用意するか、浮動小数点演算をソフトウェア的に真似する (emulation) 事になる。しかし、計算に当たって、整数と実数を一々区別するというのは面倒極まりなく、

```
real_p = real_q + int_n;
```

の様に実数・整数混合演算を書いても、コンパイラの方で適当に面倒見て呉れる様になっている。

これを更に推し進めると、複素数も混合演算したいとか、文字列も演算したい

```
("a" + "b" == "ab"; "abc" * 2 == "abcabc")
```

とか、ベクトル演算だって出来る筈だとか、欲望は果てしなく広がって行く。実際 c++ ではこうした欲望は operator overloading により全て成就可能である。そうするメリットがあればの話だが。(多くの言語が overloading している文字列代入・比較を c はあっさり無視しているが、別段デメリットになっていない)。

同じく 80x86 系 cpu では、同じ整数演算でも 2 バイト (`int`) 演算の方が 4 バイト (`long int`) 演算より効率がいいが、2 バイト演算は扱える絶対値が約 32000 と小さく、お金の計算等には不向きで、プログラマが計算の内容を考慮して `int/long int` を使い分けねばならない。

しかし、そもそも整数演算に絶対値の大きさによる本質的な違い等無いし、現に UNIX マシンや大型機では、こうした配慮無しに効率良い演算が可能で、逆に 2 バイト (`short int`) 演算の方が効率が悪い。

下手に「効率」に配慮したプログラムは、他の環境 (OS、マシン) では却って効率が悪くなったり、動かなかったり、最悪の場合間違った動きをするかも知れない。(プログラムというのは、エラーを出したり動かなかったりする方が、間違って動くよりはるかにマシである)。こうした、実行環境に依存する事柄をプログラムの中に書込んで仕舞う (hard coding) のは、そのプログラムの可搬性 portabiliby を損ない、結果的に時間とエネルギーの無駄になり、生産性を低める。

データ抽象は、プログラマの欲望の実現手段という側面もあるが、むしろプログラムの可搬性を高め、保守を容易にして、結果的にプログラミングの生産性を向上する側面の方が強調されている。

尚、こうしたデータ抽象が進むと、そのデータに対する操作に就て、正当な操作と不当なものを区別したいという欲求が生まれて自然である。

eg. `"a" * 2` は `"aa"`で正当操作。`"a" * "b"` や `"a" - 2` は不当操作。

こうした「データとその操作をセットにしたもの」をオブジェクト object と呼び、データ実体はオブジェクトの実例 instance、データの実際の操作は全てオブジェクトへのメッセージ message として扱い、実体と構造とを徹底して分離して扱うのが、今大はやりの「オブジェクト指向」Object-Oriented Paradigm(OOP、うーぷ) である。これはプログラミングに限った事ではなく、生成文法業界でも全く同じパラダイム組替えが現在進行中である。

【謝辞】

メールで御教示を賜った、Erik Naggum、奥野泰弘、土屋俊の各氏に深謝します。